# 😊 Support Vector Machine (SVM)

Support Vector Machine (SVM) is a powerful algorithm in machine learning used for classification and regression tasks, but it's mostly known for classification.

Imagine you have two different types of fruits, say apples and oranges, and you want a machine to learn how to identify them. You collect some information (data) about the fruits, like their weight and colour. Each fruit gets a label: "apple" or "orange."

Now, your goal is to create a boundary that separates apples from oranges based on their features. This is where **SVM** comes in.

**How SVM Works:**

1. **Think of it as Drawing a Line:**
    - SVM tries to draw a line (in 2D) or a plane (in 3D) that best separates the apples from the oranges. This line is called a **hyperplane**.
2. **Maximizing the Gap:**
    - SVM doesn't just draw any line, it tries to find the line that **maximizes the gap** between the two groups (apples and oranges). The idea is to create the widest possible margin between the closest apples and the closest oranges. These closest points are called **support vectors**, and they "support" the boundary.
3. **Non-linear Data:**
    - If the fruits are mixed up in a way that a straight line can't separate them, SVM can use a trick called the **kernel trick** to transform the data into a higher dimension where a separation becomes possible.

**Example:**

Let's say you have a basket of apples and oranges, and you know the weight and colour of each fruit:

- Apples: [weight = 150g, colour = red], [weight = 160g, colour = red]
- Oranges: [weight = 120g, colour = orange], [weight = 130g, colour = orange]

SVM will try to find the best way to draw a line that separates apples from oranges based on weight and colour. The goal is to make sure that when you add a new fruit with unknown features, you can easily classify it as either an apple or an orange based on which side of the line it falls on.

**Key Points:**

- **Classification:** It helps decide which category (apple or orange) a new data point belongs to.
- **Support Vectors:** These are the data points closest to the boundary and are crucial in defining the position of the line.

- **Hyperplane:** The line (or plane) that divides the categories.
- **Margin:** The distance between the hyperplane and the nearest data points from both categories.

**Real-life Examples:**

- **Spam detection:** SVM can help decide if an email is spam or not by analyzing features like keywords and frequency.
- **Face recognition:** It can classify images of faces by identifying patterns like shapes and colours.

In summary, SVM is like drawing a boundary between different groups, making sure it's as far away from the closest points of each group as possible.

**Use Cases of SVM:**

1. **Image Classification:**
    - SVM is commonly used in tasks like **face detection** or **object recognition**. For instance, it can classify images as containing a human face or not.
2. **Text Classification:**
    - SVMs are great at classifying documents, such as filtering emails into **spam** or **non-spam**. It can analyze the text features like word frequency to determine the category.
3. **Medical Diagnosis:**
    - SVMs can be used to classify medical images or data, like distinguishing between cancerous and non-cancerous cells in **tumor detection**.
4. **Handwriting Recognition:**
    - It's applied in recognizing handwritten characters or digits, like recognizing zip codes from handwritten envelopes.
5. **Bioinformatics:**
    - SVMs are used to classify biological data, such as **gene expression** data, where the goal might be to classify a gene as "active" or "inactive" based on various factors.
6. **Stock Market Prediction:**
    - SVM can help predict stock price movements by classifying whether a stock's value will rise or fall, based on historical data.

**Benefits of SVM:**

1. **Effective in High-Dimensional Spaces:**
    - SVM works well when you have a lot of features (variables). For example, in image or text data, there can be thousands of features.

2. **Good for Clear Margin of Separation:**
    - When the data points are well-separated, SVM is very efficient at creating the best boundary to classify the data.
3. **Memory Efficient:**
    - SVM uses only a subset of the data (support vectors) to define the decision boundary, which can reduce memory usage.
4. **Versatility with the Kernel Trick:**
    - SVM can be used for both linear and non-linear problems. If the data is not linearly separable (i.e., a straight line can't divide it), SVM can use a kernel to transform the data into a higher dimension where it becomes separable.
5. **Works Well in Small to Medium Datasets:**
    - SVM can give good results even with smaller datasets compared to deep learning models that require large datasets.

**Disadvantages of SVM:**

1. **Not Suitable for Large Datasets:**
    - SVM can be computationally intensive and slow when working with very large datasets, especially in training, as it involves complex optimization problems.
2. **Sensitive to Noise:**
    - SVM is sensitive to noise in the data. If the data is noisy and overlapping, it can lead to poor performance because SVM tries to find the perfect boundary, even for noisy data points.
3. **Choosing the Right Kernel is Crucial:**
    - The performance of SVM heavily depends on choosing the right kernel and its parameters. If you choose the wrong kernel, the model might underperform.
4. **Difficult to Interpret:**
    - Unlike decision trees, SVM models are harder to interpret, especially when using the kernel trick, because it's not easy to visualize or understand how the transformation works.
5. **Imbalanced Data Problems:**
    - SVM might struggle with imbalanced data (where one class has significantly more instances than another). In such cases, SVM can become biased toward the majority class.

In summary:

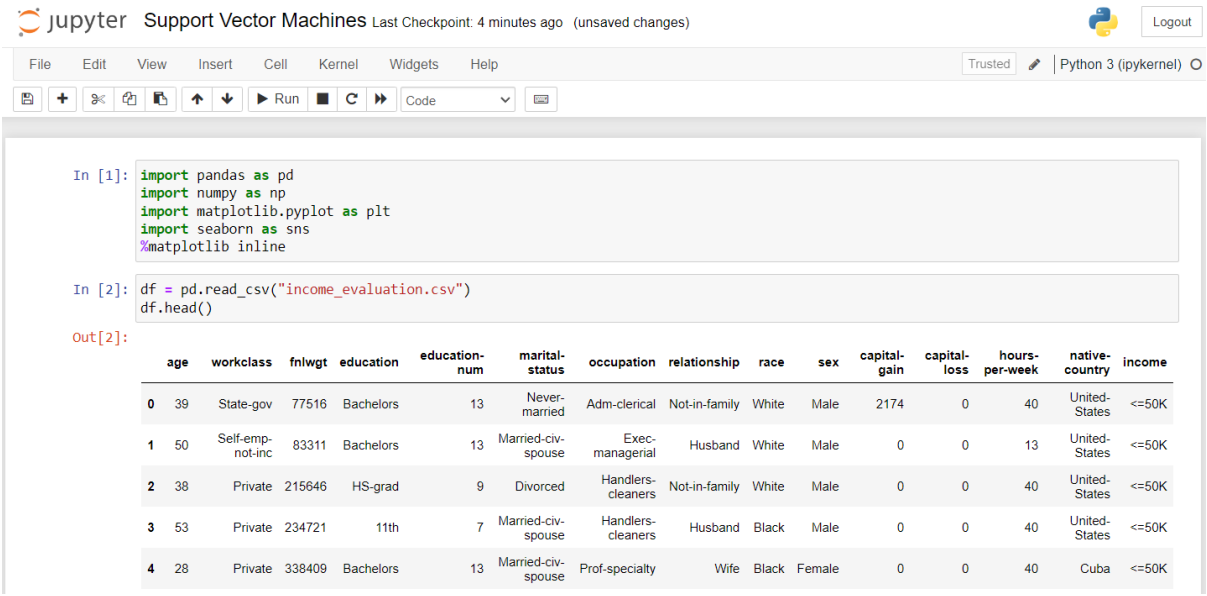
- **Use cases** include image classification, text classification, medical diagnosis, and more.

- **Benefits** include high effectiveness in high-dimensional spaces and the ability to handle non-linear data.
- **Disadvantages** include computational intensity for large datasets, sensitivity to noise, and difficulty in interpreting the model's workings.

## 😄 To begin with the Lab:

1. For this lab you need to upload income evaluation CSV file and create a new Python Kernel.


2. Then we imported some important libraries and load our dataset as data frame and displayed few entries from our dataset.




| | age | workclass | fnlwgt | education | education-num | marital-status | occupation | relationship | race | sex | capital-gain | capital-loss | hours-per-week | native-country | income |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 39 | State-gov | 77516 | Bachelors | 13 | Never-married | Adm-clerical | Not-in-family | White | Male | 2174 | 0 | 40 | United-States | <=50K |
| 1 | 50 | Self-emp-not-inc | 83311 | Bachelors | 13 | Married-civ-spouse | Exec-managerial | Husband | White | Male | 0 | 0 | 13 | United-States | <=50K |
| 2 | 38 | Private | 215646 | HS-grad | 9 | Divorced | Handlers-cleaners | Not-in-family | White | Male | 0 | 0 | 40 | United-States | <=50K |
| 3 | 53 | Private | 234721 | 11th | 7 | Married-civ-spouse | Handlers-cleaners | Husband | Black | Male | 0 | 0 | 40 | United-States | <=50K |
| 4 | 28 | Private | 338409 | Bachelors | 13 | Married-civ-spouse | Prof-specialty | Wife | Black | Female | 0 | 0 | 40 | Cuba | <=50K |

3. After that we displayed the columns and look for the values in the income column. Then we look for the unique values in the marital status column.

```
In [3]: df.columns
```

```
Out[3]: Index(['age', ' workclass', ' fnlwgt', ' education', ' education-num',
       ' marital-status', ' occupation', ' relationship', ' race', ' sex',
       ' capital-gain', ' capital-loss', ' hours-per-week', ' native-country',
       ' income'],
      dtype='object')
```

```
In [4]: df[' income'].value_counts()
```

```
Out[4]: income
        <=50K    24720
        >50K      7841
        Name: count, dtype: int64
```

```
In [5]: df[' marital-status'].unique()
```

```
Out[5]: array([' Never-married', ' Married-civ-spouse', ' Divorced',
       ' Married-spouse-absent', ' Separated', ' Married-AF-spouse',
       ' Widowed'], dtype=object)
```

4. So, what we'll do is we'll just go ahead and remove the white spaces that we have got in this data.
5. Below you can see that we have removed all the whitespaces from our dataset. We can also go ahead and adjust things as per our requirements.

```
In [6]: col_names = df.columns
        col_names = [v.strip() for v in col_names]
        col_names
```

```
Out[6]: ['age',
         'workclass',
         'fnlwgt',
         'education',
         'education-num',
         'marital-status',
         'occupation',
         'relationship',
         'race',
         'sex',
         'capital-gain',
         'capital-loss',
         'hours-per-week',
         'native-country',
         'income']
```

6. Then we ran the below command to remove the ‘fnlwgt’ column from our dataset.

```
In [7]: df.columns = col_names
        df.drop(columns="fnlwgt",inplace=True)

In [8]: df.head()
```

Out[8]:

| | age | workclass | education | education-num | marital-status | occupation | relationship | race | sex | capital-gain | capital-loss | hours-per-week | native-country | income |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 39 | State-gov | Bachelors | 13 | Never-married | Adm-clerical | Not-in-family | White | Male | 2174 | 0 | 40 | United-States | <=50K |
| 1 | 50 | Self-emp-not-inc | Bachelors | 13 | Married-civ-spouse | Exec-managerial | Husband | White | Male | 0 | 0 | 13 | United-States | <=50K |
| 2 | 38 | Private | HS-grad | 9 | Divorced | Handlers-cleaners | Not-in-family | White | Male | 0 | 0 | 40 | United-States | <=50K |
| 3 | 53 | Private | 11th | 7 | Married-civ-spouse | Handlers-cleaners | Husband | Black | Male | 0 | 0 | 40 | United-States | <=50K |
| 4 | 28 | Private | Bachelors | 13 | Married-civ-spouse | Prof-specialty | Wife | Black | Female | 0 | 0 | 40 | Cuba | <=50K |

7. By using the below command, we checked for any missing values in this dataset. But there is no missing value in this dataset.

```
In [9]: df.isnull().sum()

Out[9]: age               0
        workclass         0
        education         0
        education-num     0
        marital-status    0
        occupation        0
        relationship      0
        race              0
        sex               0
        capital-gain      0
        capital-loss      0
        hours-per-week    0
        native-country    0
        income            0
        dtype: int64
```

8. In this step, you took the **age column** (which has integer values) and converted those values into a **categorical format**. This means you transformed the continuous age data into distinct categories or groups, like dividing age ranges (e.g., 0-18 as "Young," 19-35 as "Adult," 36-60 as "Middle-aged," etc.). This pre-processing step helps make age data more useful for certain machine learning models that work better with categories rather than continuous numbers.
9. Here we performed **pre-processing** on the **Target column** (age), and in addition to that, you created a new column called **Age Type**. In this new column, you converted the numerical age values into **string format** (categorical labels), such as "Young," "Adult," or other age group categories, making the data more suitable for analysis or model training.

In [10]:
```
bins = [16,24,64,90]
labels=['young','adult','old']
df['age_types'] = pd.cut(df['age'], bins=bins,labels=labels)
df['income_num'] = np.where(df['income'] == " >50K",1,0).astype('int16')
```

In [11]:
```
df.head()
```

Out[11]:

| | age | workclass | education | education-num | marital-status | occupation | relationship | race | sex | capital-gain | capital-loss | hours-per-week | native-country | income | age_types | income_num |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 39 | State-gov | Bachelors | 13 | Never-married | Adm-clerical | Not-in-family | White | Male | 2174 | 0 | 40 | United-States | <=50K | adult | 0 |
| 1 | 50 | Self-emp-not-inc | Bachelors | 13 | Married-civ-spouse | Exec-managerial | Husband | White | Male | 0 | 0 | 13 | United-States | <=50K | adult | 0 |
| 2 | 38 | Private | HS-grad | 9 | Divorced | Handlers-cleaners | Not-in-family | White | Male | 0 | 0 | 40 | United-States | <=50K | adult | 0 |
| 3 | 53 | Private | 11th | 7 | Married-civ-spouse | Handlers-cleaners | Husband | Black | Male | 0 | 0 | 40 | United-States | <=50K | adult | 0 |
| 4 | 28 | Private | Bachelors | 13 | Married-civ-spouse | Prof-specialty | Wife | Black | Female | 0 | 0 | 40 | Cuba | <=50K | adult | 0 |

10. **Handling Missing Values**:
    - In the first three lines, you are identifying any rows where the columns **workclass**, **occupation**, or **native-country** have the value "?" (which indicates missing data). You replace these "?" values with **NaN** (which represents missing data more clearly).
    - Then, you remove any columns that contain **NaN** values using the dropna() function.
11. **Encoding Categorical Data**:
    - You use the **LabelEncoder** from scikit-learn to convert **categorical text data** (like "workclass," "education," "sex," etc.) into **numerical codes**. For example, "Private" in the workclass column might be encoded as 0, and "Self-Employed" might be encoded as 1. This makes the categorical data usable for machine learning models.
    - You repeat this encoding process for a list of columns that contain categorical values like workclass, education, occupation, etc.
12. **Scaling Numerical Data**:
    - Lastly, you use the **MinMaxScaler** to scale (normalize) all the numerical features (except for the columns **income**, **age_types**, and **income_num**) so that their values fall between 0 and 1. This helps ensure that all features contribute equally to the model, preventing large numbers from dominating the model's training process.

In short, you've cleaned the dataset by handling missing values, converted text data into numbers, and scaled the numerical data to prepare it for machine learning.

```
In [12]: df.loc[df['workclass']=='?', 'workclass']= np.NaN
         df.loc[df['occupation']=='?', 'occupation']= np.NaN
         df.loc[df['native-country']=='?', 'native_country']= np.NaN

In [13]: df = df.dropna(axis=1)

In [14]: from sklearn.preprocessing import LabelEncoder
         def label_encoder(a):
             le = LabelEncoder()
             df[a] = le.fit_transform(df[a])
         label_list = ['workclass', 'education','marital-status',
                'occupation', 'relationship', 'race', 'sex','native-country', 'income']
         for i in label_list:
             label_encoder(i)

In [15]: from sklearn.preprocessing import MinMaxScaler

In [16]: scaler = MinMaxScaler()

In [17]: scaler.fit(df.drop(['income','age_types','income_num'],axis=1))
Out[17]: MinMaxScaler
         MinMaxScaler()
```

13. **Scaling the Features**:
    - You apply the previously fitted **MinMaxScaler** to the dataset (excluding the columns **income**, **age_types**, and **income_num**) to scale the remaining numerical features. This ensures that all the numerical values are now between 0 and 1.
14. **Creating a New DataFrame**:
    - After scaling, you create a new DataFrame called **df_scaled** that stores the scaled values, and you assign the column names (like "age," "workclass," "education," etc.) to match the original dataset's columns.
15. **Viewing the First Few Rows**:
    - The last line (df_scaled.head()) displays the first few rows of the new scaled DataFrame, allowing you to see the transformed and scaled values for the selected features.

In short, you've scaled your numerical data and created a new DataFrame to store and view the scaled features.

```
In [18]: scaled_features = scaler.transform(df.drop(['income','age_types','income_num'],axis=1))

In [19]: columns=['age', 'workclass', 'education', 'education_num', 'marital_status',
                'occupation', 'relationship', 'race', 'sex', 'capital_gain',
                'capital_loss', 'hours_per_week', 'native_country']

In [20]: df_scaled = pd.DataFrame(scaled_features,columns=columns)
         df_scaled.head()
```

Out[20]:

| | age | workclass | education | education_num | marital_status | occupation | relationship | race | sex | capital_gain | capital_loss | hours_per_week | native_countr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0.301370 | 0.875 | 0.600000 | 0.800000 | 0.666667 | 0.071429 | 0.2 | 1.0 | 1.0 | 0.02174 | 0.0 | 0.397959 | 0.9512 |
| 1 | 0.452055 | 0.750 | 0.600000 | 0.800000 | 0.333333 | 0.285714 | 0.0 | 1.0 | 1.0 | 0.00000 | 0.0 | 0.122449 | 0.9512 |
| 2 | 0.287671 | 0.500 | 0.733333 | 0.533333 | 0.000000 | 0.428571 | 0.2 | 1.0 | 1.0 | 0.00000 | 0.0 | 0.397959 | 0.9512 |
| 3 | 0.493151 | 0.500 | 0.066667 | 0.400000 | 0.333333 | 0.428571 | 0.0 | 0.5 | 1.0 | 0.00000 | 0.0 | 0.397959 | 0.9512 |
| 4 | 0.150685 | 0.500 | 0.600000 | 0.800000 | 0.333333 | 0.714286 | 1.0 | 0.5 | 0.0 | 0.00000 | 0.0 | 0.397959 | 0.1219 |

16. **Handling Imbalanced Data with SMOTETomek**:
    - The dataset has an **imbalance** problem, meaning that one class (e.g., "income" column categories) has more samples than the other.
    - To fix this, you're using **SMOTETomek**, which is a combination of oversampling (adding more samples of the minority class) and undersampling (removing some samples from the majority class) to create a balanced dataset. This step helps the model learn from both classes equally.
    - After applying **SMOTETomek**, you get new balanced feature and label sets: **X_res** (features) and **y_res** (labels).
17. **Splitting the Data into Training and Test Sets**:
    - You split the resampled data into two parts: **80% for training** the model and **20% for testing** the model, using the train_test_split function. The shuffle=True option ensures that the data is randomly mixed before splitting.
18. **Training a Support Vector Machine (SVM) Model**:
    - You create an **SVM classifier** (SVC), which is a type of machine learning model used for classification tasks.
    - Using **cross-validation**, you train the SVM model on different parts of the training data (split into 5 subsets), calculating accuracy for each fold. This ensures that the model generalizes well.
    - Then, you fit the SVM model on the **full training data** using svc.fit().
19. **Evaluating Model Performance**:
    - You calculate and print the average training accuracy (Train Score) based on the cross-validation results.
    - You also calculate the **test accuracy** (Test Score) by evaluating the model's performance on the 20% test data.

    In summary, you handled imbalanced data using SMOTETomek, split the data into training and test sets, trained an SVM model, and then evaluated its performance on both the training and test sets to see how well it learned.
20. The results indicate the performance of the Support Vector Machine (SVM) model:
    - **Train Score**: 0.83 (or **83%**), meaning that during training, the model was able to correctly classify about 83% of the examples based on cross-validation.
    - **Test Score**: 0.83 (or **83%**), meaning the model correctly classified about 83% of the unseen test data.

    This suggests that the model is performing consistently well on both the training and test data, indicating that it has learned to generalize without overfitting.

In [21]:
```
from imblearn.combine import SMOTETomek
from imblearn.under_sampling import NearMiss

X = df_scaled
y= df.income

# Implementing Oversampling for Handling Imbalanced
smk = SMOTETomek(random_state=42)
X_res,y_res=smk.fit_resample(X,y)

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_res,y_res,test_size=0.20,random_state=101,shuffle=True)
from sklearn.model_selection import cross_val_score

from sklearn.svm import SVC
svc = SVC(random_state = 101)
accuracies = cross_val_score(svc, X_train, y_train, cv=5)
svc.fit(X_train,y_train)

print("Train Score:",np.mean(accuracies))
print("Test Score:",svc.score(X_test,y_test))
```

```
Train Score: 0.8303309471891493
Test Score: 0.8333679401993356
```