# 😊 Logistic Regression & EDA

Logistic Regression is a simple machine learning method used to make decisions between two possible outcomes, like "yes" or "no", "0" or "1". It helps us figure out which category something belongs to based on the information we have.

Let's break it down in an easy way:

**What does it do?**

Imagine you're a teacher trying to predict if a student will pass or fail an exam. You know things like how many hours they studied, their past grades, and how focused they are in class. Logistic Regression helps you **predict** the outcome (pass/fail) based on these details (study hours, grades, focus).

**How does it work?**

Logistic Regression looks at the **input** data (like hours studied, past grades) and gives a **probability** that something will happen (like passing or failing). For example:

- If the probability is 0.8 (80%), we can say the student is likely to **pass**.
- If the probability is 0.3 (30%), we can say the student is likely to **fail**.

The model draws a line (called a decision boundary) to split the possible outcomes into "pass" and "fail" based on the input data.

**The Sigmoid Function (in simple terms)**

The main thing that logistic regression uses is something called a **sigmoid function**. All this function does is take any input and squeeze it into a value between 0 and 1 (like a probability):

- 0 means "unlikely" or "no"
- 1 means "very likely" or "yes"

For example, if the model calculates a value of 0.7, it interprets that as a 70% chance of passing. If it calculates 0.4, that's a 40% chance of passing.

**Example 1: Predicting Exam Results**

Let's say you want to predict whether students will pass or fail an exam based on how many hours they studied. You collect data like this:

| Hours Studied | Result (Pass/Fail) |
|---|---|
| 5 | Pass |
| 1 | Fail |
| 7 | Pass |
| 2 | Fail |

Logistic Regression will look at this data and try to figure out the relationship between the hours studied and the chance of passing. After learning from this data, it might predict that:

- If a student studied 6 hours, they have an 85% chance of passing.
- If a student studied only 1 hour, they have a 20% chance of passing.

**Example 2: Spam Email Detection**

Another easy example is **spam detection** in emails:

- The input could be things like words in the email ("Free", "Prize", etc.).
- The output is whether the email is **spam (1)** or **not spam (0)**.

Logistic Regression looks at patterns in the emails and learns what kind of content is more likely to be spam. Then, when a new email comes in, it gives a probability of it being spam (e.g., 80%) or not spam (e.g., 20%).

**In Summary:**

- Logistic Regression helps you make decisions between **two options**.
- It gives you a **probability** of something happening, like passing an exam or being spam.
- It uses past data to learn patterns, and then predicts future outcomes.

## 😄 Gradient Descent

Gradient Descent is a method in machine learning that helps a model **learn** by finding the best possible values for the model's parameters (like weights). It's used to minimize (or reduce) the errors a model makes in predictions.

**Think of it like this:**

Imagine you're standing at the top of a hill (this represents a model with bad predictions). Your goal is to get to the bottom of the hill, which is the **lowest point** where your model makes the **least amount of error**. But, the catch is, you're **blindfolded**, so you can't see where the bottom is.

You need to **feel your way down** step by step, taking small steps downhill to reach the lowest point. That's essentially what **gradient descent** does in machine learning—it takes small steps towards reducing the error until the model is as accurate as possible.

**Here's a breakdown:**

1. **Starting Point**: Imagine you start with some random guesses (random values for your model's weights or parameters). This is like standing at some random spot on the hill.
2. **Slope (Gradient)**: To figure out which way to go, you "feel" which direction the ground is sloping. If it's sloping down, you take a step in that direction. In gradient descent, the slope (or gradient) tells the model which direction to move the parameters to reduce the error. If the slope is steep, it means there's a big error to reduce.

3. **Taking Steps (Learning Rate)**: Gradient descent takes steps downhill. The size of each step is controlled by something called the **learning rate**:
    - If the step is too **big**, you might miss the lowest point and overshoot.
    - If the step is too **small**, it will take too long to reach the bottom.
    - The key is to find the right size of steps to make steady progress.
4. **Reaching the Bottom**: As you keep stepping down, you eventually reach the lowest point on the hill—this is where the model's error is minimized (called **convergence**). Here, the model has the best possible parameters.

**Example: Baking a Cake**

Let's say you're trying to bake the perfect cake, but you're experimenting with the amount of sugar to use. Each time you bake the cake, you get feedback (let's say a score out of 10).

Your goal is to find the right amount of sugar to get the best score (like finding the lowest error in gradient descent).

Here's how it relates:

- **Random start**: The first time, you might randomly choose 300g of sugar.
- **Feedback (slope)**: You bake the cake and it's too sweet (score: 3/10). This feedback tells you the cake needs less sugar (the slope tells you to go in the opposite direction).
- **Adjust (step)**: So next time, you reduce the sugar to 200g. Now the score improves (6/10), meaning you're moving in the right direction.
- **Keep adjusting**: You keep reducing or increasing sugar, taking smaller steps as you get closer to the perfect amount, aiming for a score of 10/10.

**How Gradient Descent Works in ML:**

- The algorithm tries different values for the model's parameters (like how much "sugar" to use).
- It gets feedback on how wrong the predictions are (like the cake's score).
- It adjusts the parameters step by step (like adjusting sugar), to minimize the error (make a better cake).

**Visualizing Gradient Descent:**

Imagine a **bowl**—the top edges of the bowl represent high errors, and the bottom of the bowl represents the least error (best possible model). Gradient descent is like rolling a ball down the sides of the bowl to reach the bottom.

**Summary in Layman's Terms:**

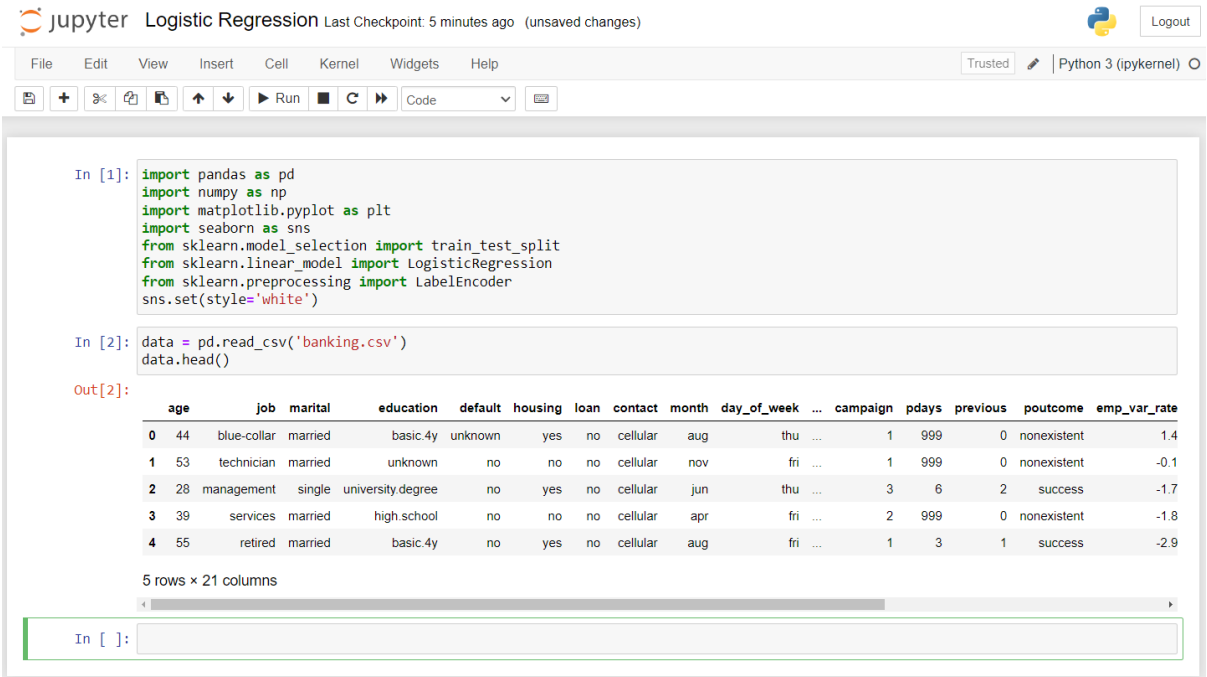
- **Gradient Descent** is a way for a machine learning model to improve itself by taking small steps to reduce errors.

- It’s like walking downhill blindfolded—you feel the slope and take steps until you reach the lowest point (least error).
- The **learning rate** controls the size of your steps, and the model keeps adjusting its values until it’s making the best predictions.

## 😄 To begin with the Lab:

1. For this lab, you need to upload the banking.csv file and create a new Python Kernel.


2. Then we imported some important libraries, loaded our dataset as a data frame, and displayed the first few entries.




3. After that we displayed the columns and information of our data frame.

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

```
Out[3]: Index(['age', 'job', 'marital', 'education', 'default', 'housing', 'loan',
               'contact', 'month', 'day_of_week', 'duration', 'campaign', 'pdays',
               'previous', 'poutcome', 'emp_var_rate', 'cons_price_idx',
               'cons_conf_idx', 'euribor3m', 'nr_employed', 'y'],
              dtype='object')
```

```
In [4]: data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 41188 entries, 0 to 41187
Data columns (total 21 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   age             41188 non-null  int64
 1   job             41188 non-null  object
 2   marital         41188 non-null  object
 3   education       41188 non-null  object
 4   default         41188 non-null  object
 5   housing         41188 non-null  object
 6   loan            41188 non-null  object
 7   contact         41188 non-null  object
 8   month           41188 non-null  object
 9   day_of_week     41188 non-null  object
 10  duration        41188 non-null  int64
 11  campaign        41188 non-null  int64
 12  pdays           41188 non-null  int64
 13  previous        41188 non-null  int64
 14  poutcome        41188 non-null  object
```

4. Here we have displayed the loan column, in a similar way you can display the other columns just replace the column name with any of the column of your choice.

```
In [5]: data['loan'].value_counts()
```

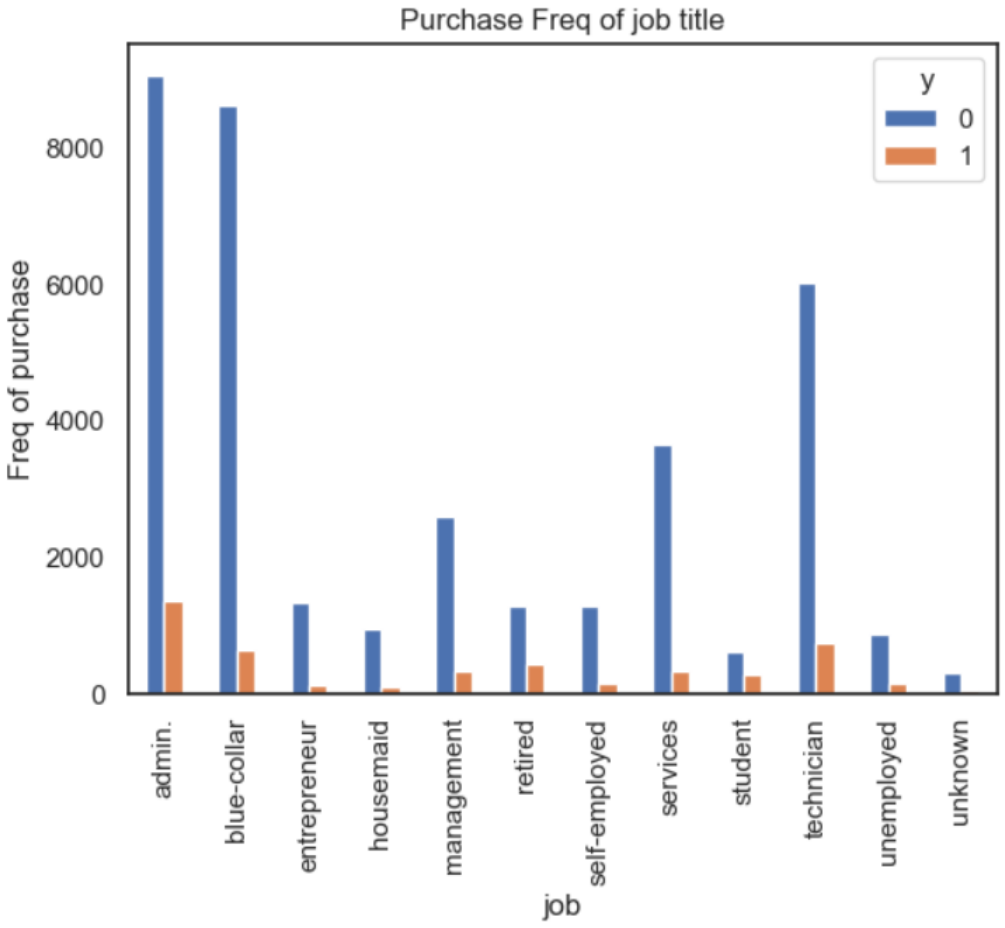
```
Out[5]: loan
        no         33950
        yes         6248
        unknown      990
        Name: count, dtype: int64
```

5. When you look at the data in the column called "Y", you're checking how many times certain values appear. You found that:
   The value **0** shows up **3,654** times.
   The value **1** shows up **4,640** times.
   So, you have two different values in "Y", and you count how many rows have each of those values.
6. Here we can see that there is an imbalance of the class.

```
In [6]: data['y'].value_counts()

Out[6]: y
        0    36548
        1     4640
        Name: count, dtype: int64
```

7. Next, we want to **understand the relationship** between how often people in a specific job title make purchases.
8. To compare this, you'll use something called a **crosstab** (short for cross-tabulation), which helps organize and compare data for different job titles.
9. After that, you'll create a **bar chart** to visually display the results, making it easier to see how different job titles relate to the purchase frequency.

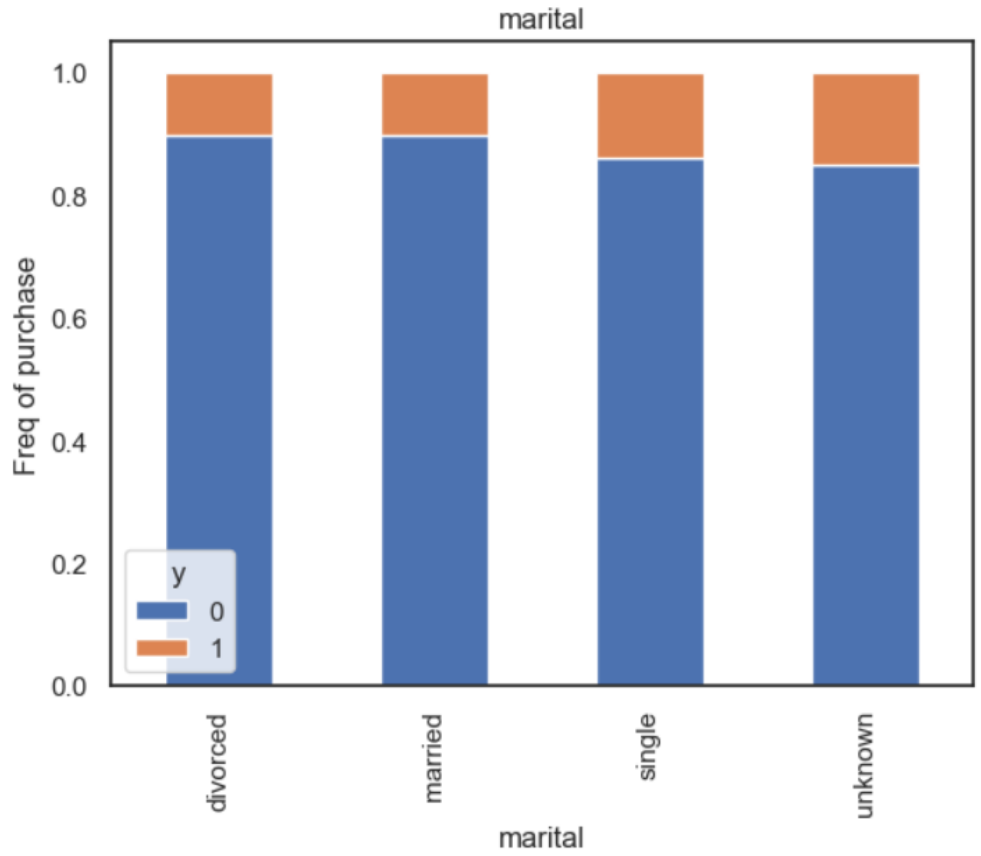
```
In [9]: %matplotlib inline
        pd.crosstab(data['job'],data['y']).plot(kind='bar')
        plt.title("Purchase Freq of job title")
        plt.xlabel("job")
        plt.ylabel('Freq of purchase')

Out[9]: Text(0, 0.5, 'Freq of purchase')
```



10. Now, let's see if being **married** has any effect on whether a person is likely to **buy the product** or not.
11. In other words, we want to check if a person's marital status influences their decision to make a purchase.

```
In [10]: %matplotlib inline
         table = pd.crosstab(data['marital'],data['y'])
         table.div(table.sum(1).astype(float), axis=0).plot(kind='bar', stacked=True)
         plt.title("marital")
         plt.xlabel("marital")
         plt.ylabel('Freq of purchase')
```

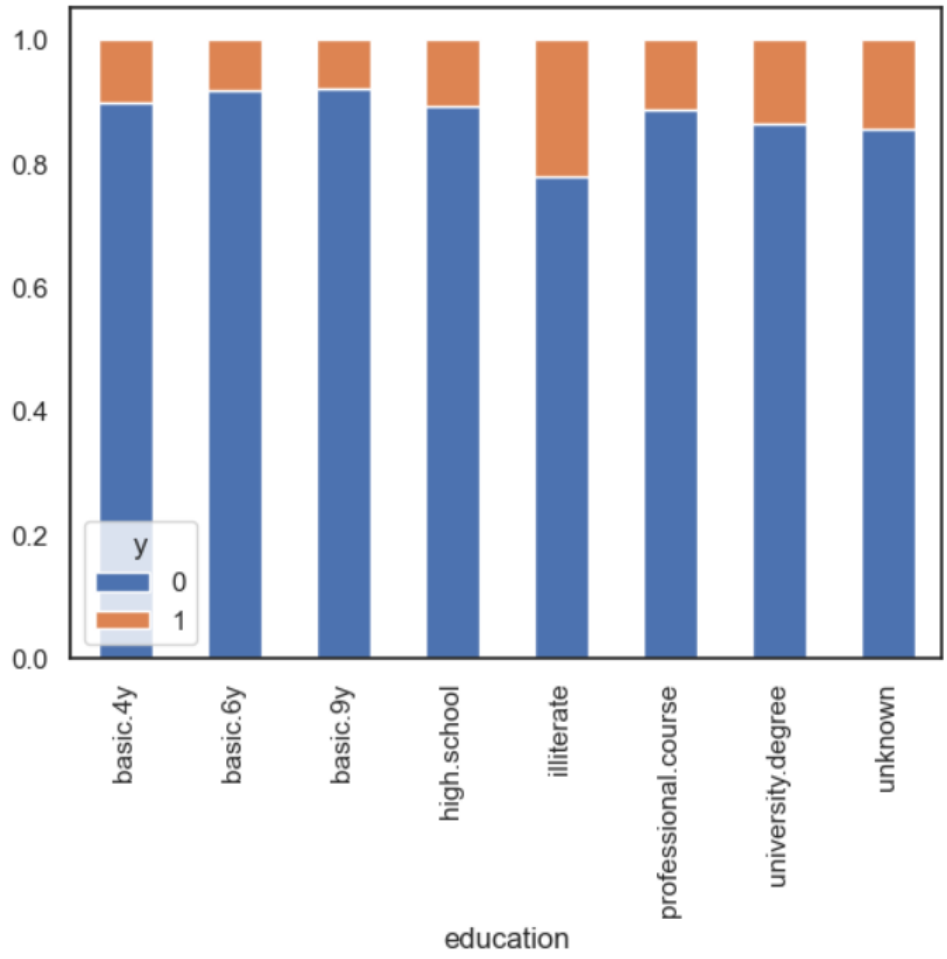
```
Out[10]: Text(0, 0.5, 'Freq of purchase')
```




12. Next, we'll also create a **visualization** to see the data clearly. Now, let's move forward and this time, we'll look at the impact of **education** on whether someone buys the product or not.

In [11]:
```
%matplotlib inline
table = pd.crosstab(data['education'],data['y'])
table.div(table.sum(1).astype(float), axis=0).plot(kind='bar', stacked=True)
```

Out[11]: <Axes: xlabel='education'>




13. Now, let's move on and **analyze** one of the **numerical columns**. For this, we'll look at the **age** column to see how a person's age might affect their behavior or decisions, like making a purchase.

In [12]:
```
data.head()
```

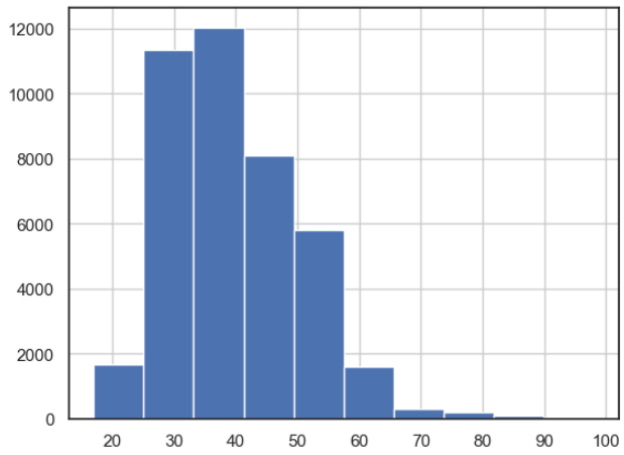
Out[12]:

| | age | job | marital | education | default | housing | loan | contact | month | day_of_week | ... | campaign | pdays | previous | poutcome | emp_var_rate |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 44 | blue-collar | married | basic.4y | unknown | yes | no | cellular | aug | thu | ... | 1 | 999 | 0 | nonexistent | 1.4 |
| 1 | 53 | technician | married | unknown | no | no | no | cellular | nov | fri | ... | 1 | 999 | 0 | nonexistent | -0.1 |
| 2 | 28 | management | single | university.degree | no | yes | no | cellular | jun | thu | ... | 3 | 6 | 2 | success | -1.7 |
| 3 | 39 | services | married | high.school | no | no | no | cellular | apr | fri | ... | 2 | 999 | 0 | nonexistent | -1.8 |
| 4 | 55 | retired | married | basic.4y | no | yes | no | cellular | aug | fri | ... | 1 | 3 | 1 | success | -2.9 |

5 rows × 21 columns

In [13]:
```
data['age'].hist()
```

Out[13]: <Axes: >




14. Now, what I'm going to do is create **dummy variables** for the data set.
15. I'm using the **get_dummies** function because a lot of the data we have is **nominal** (i.e., it's categorized but doesn't have a specific order, like job titles or marital status).
16. Since the data is nominal, and as we discussed earlier in **Exploratory Data Analysis (EDA)**, the right technique to handle this is called **one-hot encoding**.
17. The easiest way to do one-hot encoding is by using the **get_dummies** function, which will convert the categories into numerical values that the model can understand.
18. Now we have all the data in **numerical format**. Any data that was originally in **string format** has been converted into numbers using **one-hot encoding**. This makes it easier to work with and analyze in our model.

In [14]:
```
data_final = pd.get_dummies(data, drop_first=True)
data_final.head()
```

Out[14]:

| | age | duration | campaign | pdays | previous | emp_var_rate | cons_price_idx | cons_conf_idx | euribor3m | nr_employed | ... | month_may | month_nov | month_oct |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 44 | 210 | 1 | 999 | 0 | 1.4 | 93.444 | -36.1 | 4.963 | 5228.1 | ... | False | False | False |
| 1 | 53 | 138 | 1 | 999 | 0 | -0.1 | 93.200 | -42.0 | 4.021 | 5195.8 | ... | False | True | False |
| 2 | 28 | 339 | 3 | 6 | 2 | -1.7 | 94.055 | -39.8 | 0.729 | 4991.6 | ... | False | False | False |
| 3 | 39 | 185 | 2 | 999 | 0 | -1.8 | 93.075 | -47.1 | 1.405 | 5099.1 | ... | False | False | False |
| 4 | 55 | 137 | 1 | 3 | 1 | -2.9 | 92.201 | -31.4 | 0.869 | 5076.2 | ... | False | False | False |

5 rows × 54 columns

19. Since we have **imbalanced data** (where one class has significantly more samples than another), we will use **SMOTE** (Synthetic Minority Over-sampling Technique) to balance the dataset.
20. This technique helps create more examples of the underrepresented class, making the dataset more balanced for better analysis and modeling.

In [15]:
```
# Smote
X = data_final.drop(columns='y')
y = data_final['y']
```

In [16]:
```
from imblearn.over_sampling import SMOTE
os = SMOTE(random_state=0)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
columns = X_train.columns

os_data_X,os_data_y=os.fit_resample(X_train, y_train)
os_data_X = pd.DataFrame(data=os_data_X,columns=columns )
os_data_y= pd.DataFrame(data=os_data_y,columns=['y'])
```

In [17]:
```
os_data_y['y'].value_counts(normalize=True)
```

Out[17]:
```
y
0    0.5
1    0.5
Name: proportion, dtype: float64
```

21. Now, we are going to implement a common technique in machine learning called **Recursive Feature Elimination (RFE)**.
22. This technique will be used with our **logistic regression algorithm**.
23. Recursive Feature Elimination works by repeatedly building a model and selecting either the best or the worst-performing feature. After selecting a feature, we set it aside and repeat the process with the remaining features.
24. We continue this process until we've evaluated all the features in the dataset.
25. The main goal of RFE is to select the most important features by gradually considering smaller and smaller sets of features.
26. As a result of this process, I ran my **logistic regression model** and specified that I wanted to select **20 features**.
27. Now, it has identified and shortlisted those 20 important features.
28. If you want to see the list of features it selected, here they are. These are the columns that are considered important for understanding the relationship between **X** (the input features) and **Y** (the target variable).

In [18]:
```
# Recursive Feature Elimination technique
from sklearn.feature_selection import RFE

logreg = LogisticRegression()
rfe = RFE(logreg, n_features_to_select=20)
rfe = rfe.fit(os_data_X, os_data_y.values.ravel())
print(rfe.support_)
print(rfe.ranking_)
```

```
  n_iter_i = _check_optimize_result(

C:\Users\PULKIT\anaconda3\Lib\site-packages\sklearn\linear_model\_logistic.py:458: ConvergenceWarning: lbfgs failed to converg
e (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(

[False False False False False False False False False False False False
 False False  True False False False False False  True  True  True
  True  True  True  True  True  True  True False False False False False
 False False False  True False False  True False False  True  True  True
  True  True  True False  True]
[31 30 26 33 25 27 28 32 18 29  9 12  7 16  1 14 13  8 15 10 11  1  1  1
  1  1  1  1  1 22 34 19 20 24 21 23  5  1  3  2  1  6  4  1  1  1
  1  1  1 17  1]
```

```
In [19]: cols = X.columns[rfe.support_]
         cols

Out[19]: Index(['job_retired', 'marital_married', 'marital_single', 'marital_unknown',
                'education_basic.6y', 'education_basic.9y', 'education_high.school',
                'education_illiterate', 'education_professional.course',
                'education_university.degree', 'education_unknown', 'month_dec',
                'month_mar', 'month_oct', 'month_sep', 'day_of_week_mon',
                'day_of_week_thu', 'day_of_week_tue', 'day_of_week_wed',
                'poutcome_success'],
               dtype='object')
```

29. What I'll do now is create a new variable called **columns** to store all the selected feature names.
30. Next, I'll use only these selected features for my **X** (input data) and **Y** (target variable).
31. Once I've selected these features for **X** and **Y**, we can proceed to **fit** the model using this dataset.

```
In [20]: X=os_data_X[cols]
         y=os_data_y['y']

In [21]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
         logreg = LogisticRegression()
         logreg.fit(X_train, y_train)

Out[21]: LogisticRegression
         LogisticRegression()

In [22]: y_pred = logreg.predict(X_test)
         print('Accuracy of logistic regression classifier on test set: {:.2f}'.format(logreg.score(X_test, y_test)))

         Accuracy of logistic regression classifier on test set: 0.89
```

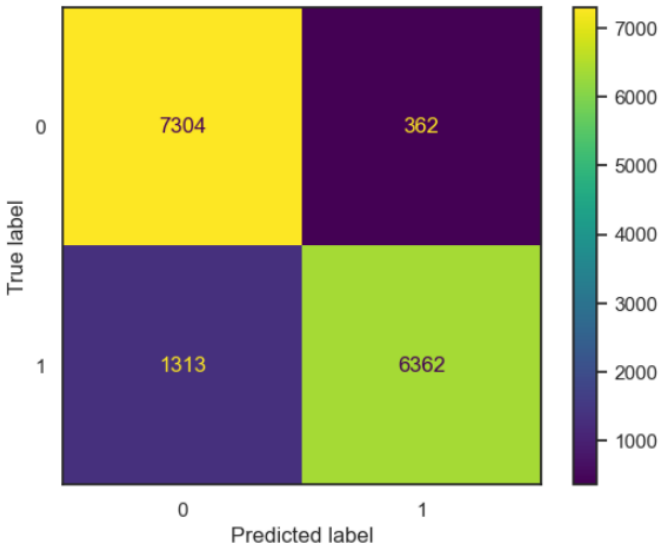
32. First, I will generate the **confusion matrix**.
33. To do this, I'll import the **confusion_matrix** function from the **sklearn.metrics** module.
34. This will help us evaluate the performance of the model by comparing the actual and predicted values.

## Evaluation metrics of Classification task

```
In [ ]: # accuracy --> number of correct predictions/total number of predictions
```

```
In [23]: from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix, f1_score, precision_score,recall_score
```

```
In [24]: cm = confusion_matrix(y_test,y_pred)
         cm_display = ConfusionMatrixDisplay(cm, display_labels=[0,1])
         cm_display.plot()
         plt.show()
```




- True Negative (Top-Left Quadrant)
- False Positive (Top-Right Quadrant)
- False Negative (Bottom-Left Quadrant)
- True Positive (Bottom-Right Quadrant)

35. Here's a clearer version of what you're saying: Now, from the **confusion matrix**, we can directly calculate the **precision score, recall score and f1 score**.
36. I'm going to print the precision score using the appropriate function to see how accurately the model predicts the positive class.

```
In [25]: print(f"Precision score is {precision_score(y_test,y_pred)}")
         print(f"Recall score is {recall_score(y_test,y_pred)}")
         print(f"f1-score is {f1_score(y_test,y_pred)}")
```

```
Precision score is 0.946162998215348
Recall score is 0.8289250814332247
f1-score is 0.8836724772553648
```

37. In addition to these metrics, there's another important metric used in classification called the **ROC AUC score**.

In [26]:
```
from sklearn.metrics import roc_auc_score
y_pred_proba = logreg.predict_proba(X_test)
y_pred_proba
```

Out[26]:
```
array([[1.62804980e-03, 9.98371950e-01],
       [2.20863983e-04, 9.99779136e-01],
       [7.39736030e-01, 2.60263970e-01],
       ...,
       [1.63807256e-01, 8.36192744e-01],
       [1.19856655e-02, 9.88014335e-01],
       [2.26356147e-01, 7.73643853e-01]])
```

In [27]:
```
roc_auc_score(y_test,y_pred_proba[:,1])
```

Out[27]:
```
0.9253646415365958
```

In [28]:
```
from sklearn.metrics import roc_curve
```

In [29]:
```
fpr,tpr, threshoold = roc_curve(y_test,y_pred_proba[:,1])
random_probs = [0 for i in range(len(y_test))]
p_fpr, p_tpr, _ = roc_curve(y_test, random_probs, pos_label=1)
plt.plot(fpr, tpr, linestyle='--',color='orange', label='Logistic Regression')
plt.plot(p_fpr, p_tpr, linestyle='--', color='blue')
plt.title('ROC curve')
# x label
plt.xlabel('False Positive Rate')
# y label
plt.ylabel('True Positive rate')
plt.legend(loc='best')
plt.show();
```