DATA SOCIETY®

Advanced classification - logistic regresssion

"One should look for what is and not what he thinks should be."
-Albert Einstein.

Who we are

- Data Society's mission is to **integrate Big Data and machine learning best practices across entire teams** and empower professionals to identify new insights
- We provide:
  - High-quality data science training programs
  - Customized executive workshops
  - Custom software solutions and consulting services
- Since 2014, we've worked with thousands of professionals to make their data work for them

Using Zoom

Raise your hand if you can hear me OK.

- From the toolbar (probably on the bottom of your screen), ***select the button marked "Reactions" and choose the hand***

In the chat box, tell everyone what you see out the nearest window.

- From the toolbar (probably on the bottom of your screen), ***select the button marked "Chat."*** The chat box should appear. On smaller screens, the "Chat" button may be hiding under the "More" menu.

Best practices for virtual classes

1. Find a quiet place, free of as many distractions as possible. Headphones are recommended.
2. Stay on mute unless you are speaking.
3. Remove or silence alerts from cell phones, e-mail pop-ups, etc.
4. Participate in activities and ask questions. This will be interactive!
5. Give your honest feedback so we can troubleshoot problems and improve the course.

Getting to know your classmates

- Let's head into a breakout room and introduce ourselves
- You'll have 5-10 minutes to exchange your names and departments and talk about what problems you hope to solve by taking this course
- When you come back, be ready to share 1-2 topic areas that came up as project interests with the whole group!

Getting started: data scientists

- Data scientists are analytical data experts that have the technical skills to solve complex problems
- They can:

1. **Pose** the right question
2. **Wrangle** data (gather, clean, and sample data to get a suitable dataset)
3. **Manage** data for easy access by the organization
4. **Explore** data to generate a hypothesis
5. **Make predictions** using statistical methods such as regression and classification
6. **Communicate** results using visualizations, presentations, and products

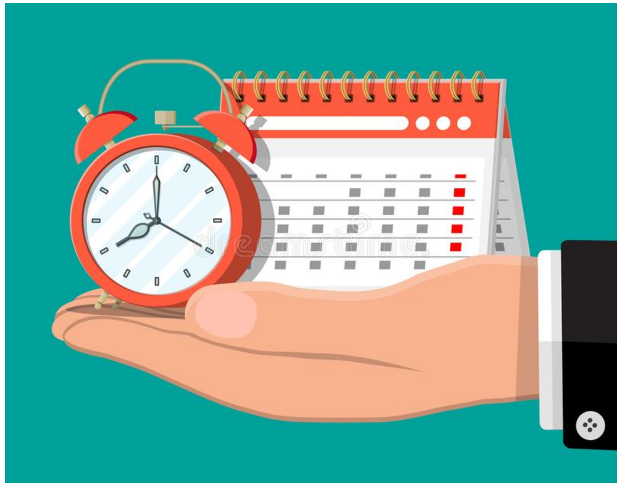
Getting started: your proficiency

- You don't need to be a data scientist to have programming as part of your professional toolkit
- The level of proficiency you can achieve will depend on:
  - **the problems** you are trying to solve on daily basis
  - **the subject matter area** you are in
  - **the level of complexity** your programming solution demands

Getting started: the data science control cycle

No matter your level of proficiency, it's important to be familiar with the data science control cycle

Getting started: how we teach



- We'll walk through the concepts and code together, then you'll have the opportunity to answer questions and practice
- You should have the following:
  - Code files to follow along with the slides
  - Links to interactive knowledge checks
  - Exercise files (we give you 2 files and one has the answers)
- Recordings will be made available

Getting started: Python

- In this course, we will be using the **Python** programming language, as well as a few helper packages that you should have already installed:
  - OS
  - NumPy
  - Pandas
  - Matplotlib
  - Pickle
  - Scikit-learn
- Let's start by preparing our environment

Getting started: directory settings

- In order to maximize the efficiency of the workflow, we encode the directory structure into `variables`
- Let the `main_dir` be the variable corresponding to your `advanced-classification` folder

```
from pathlib import Path
# Set `home_dir` to the root directory of your computer.
home_dir = Path.home()

# Set `main_dir` to the location of your `advanced-classification` folder.
main_dir = home_dir / "Desktop" / "advanced-classification"

# Make `data_dir` from the `main_dir` and remainder of the path to data directory.
data_dir = main_dir / "data"
```

Getting started: working directory

- Set the working directory to `data_dir`

```
# Set working directory.
os.chdir(data_dir)
```

```
# Check working directory.
print(os.getcwd())
```

```
/home/[user-name]/Desktop/advanced-classification/data
```

Getting started: loading packages

- Load the packages we will be using

```
# Helper packages.
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import pickle
```

```
# Scikit-learn package for logistic regression.
from sklearn import linear_model
```

```
# Model set up and tuning packages from scikit-learn.
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
```

```
# Scikit-learn packages for evaluating model performance.
from sklearn import metrics
```

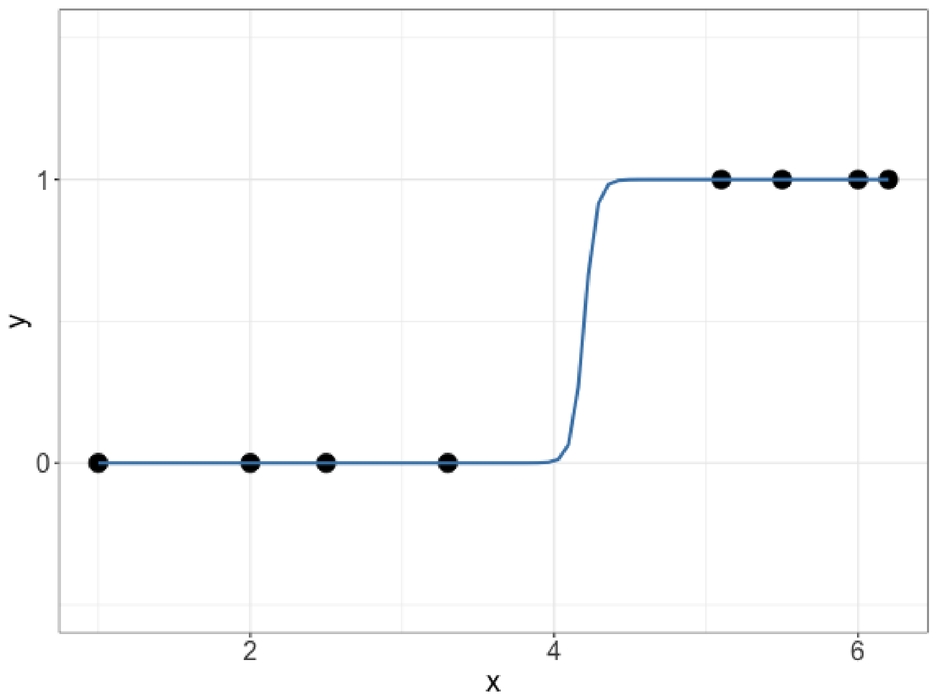
```
# Scikit-learn package for data preprocessing.
from sklearn import preprocessing
```

Module completion checklist

Objective	Complete
Determine when to use logistic regression for classification and transformation of target variable	
Summarize the process and the math behind logistic regression	
Implement logistic regression on a training dataset and predict on test	
Review classification performance metrics and assess results of logistic model performance	
Transform categorical variables for implementation of logistic regression	
Implement logistic regression on the data and assess results of classification model performance	
Analyze the model to determine if / when overfitting occurs	
Demonstrate tuning the model using grid search cross-validation	

Logistic regression: what is it?

- **Supervised** machine learning method
- Target/dependent variable is **binary** (one/zero)
- Outputs the **probability** that an observation will be in the desired class (`y = 1`)
- Solves for coefficients to create a *curved* function to maximize the likelihood of correct classification
- `logistic` comes from the `logit` function (*a.k.a. sigmoid function*)



Logistic regression: when to use it?

- We use it to classify data into **categories**
  - Is a given email **spam** or **not spam?**
- It outputs **probabilities**, not actual class labels
  - Easily tweak its performance by adjusting a **cut-off probability**
  - No need to re-run the model with new parameters
- It is a **well-established algorithm**
  - It has implementations **across many programming languages**
  - We can create robust, efficient, and well-optimized models

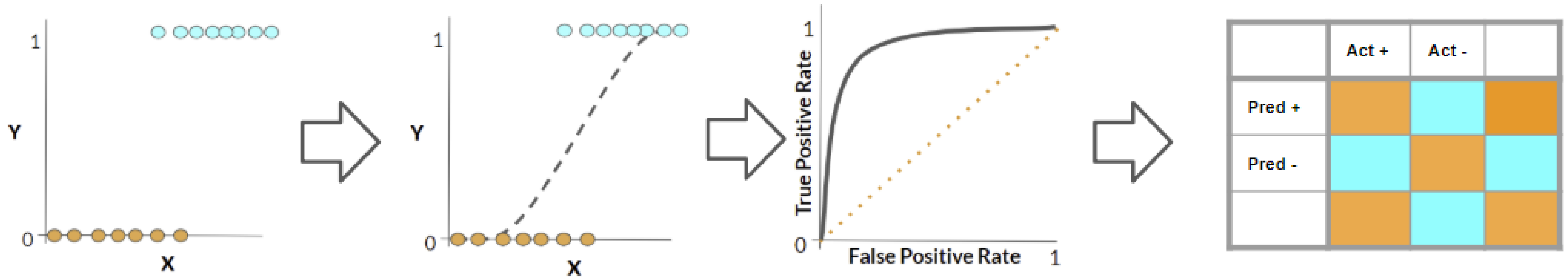
Logistic regression: process

Step 1:
Convert target variable to 1/0

Step 2:
Logistic regression on training data

Step 3:
Use ROC curve & AUC to pick threshold

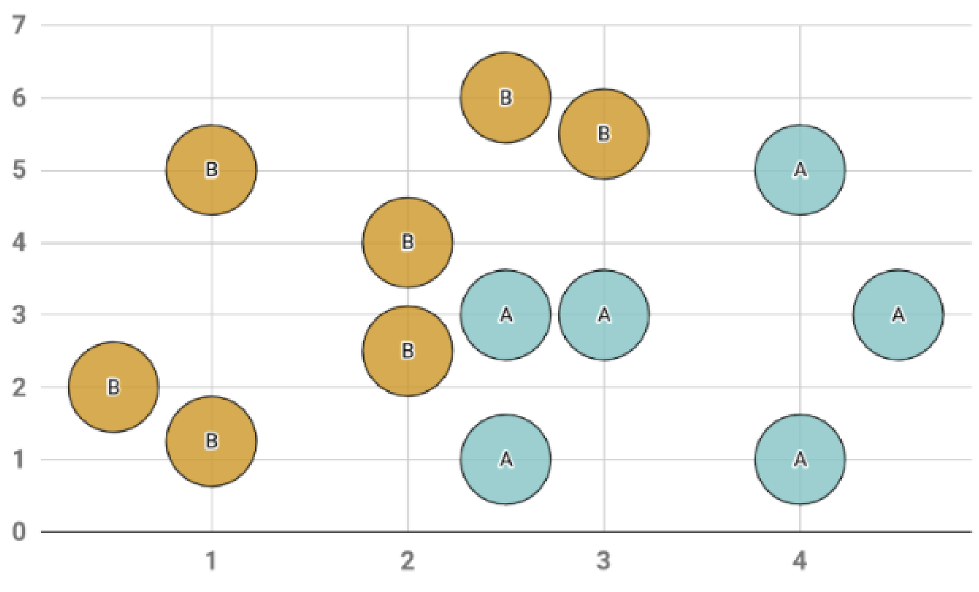
Step 4:
Check performance on test data

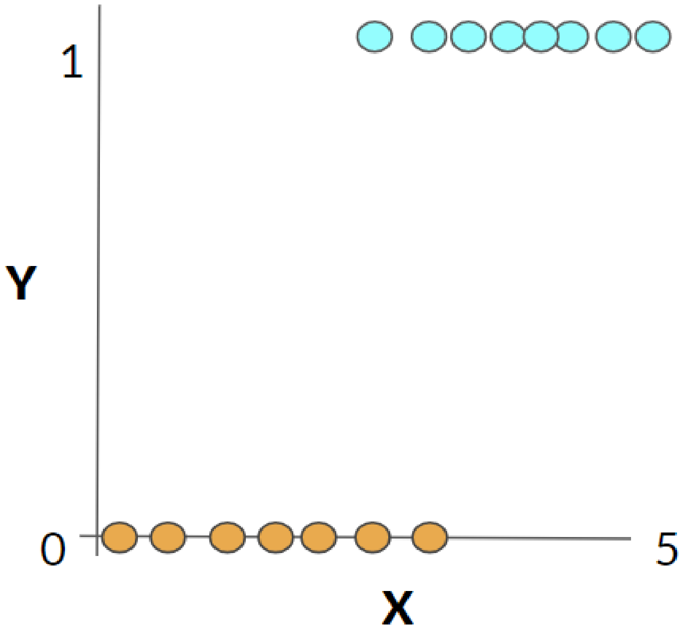


Categorical to binary target variable

Two main ways to prepare the target variable:

- **First method:** translate an existing binary variable (i.e. spam/not spam) into `1` and `0`





But what if we have **continuous** data, without defined categories?

Continuous to binary target variable

- **Second method:** convert a continuous numeric variable into binary one
  - We can do this by using a threshold and labeling observations that are higher than that threshold as `1` and `0` otherwise
  - Let's say we have a column `charge` which indicates the cost of a product
  - If the median (threshold) for the variable `charge` below was `100`, then any point below the median is `0`, and any point above is `1`

Charge
193.89
0
39.99
201.65
117.9
200.88
79.99

Charge
1
0
0
1
1
1
0

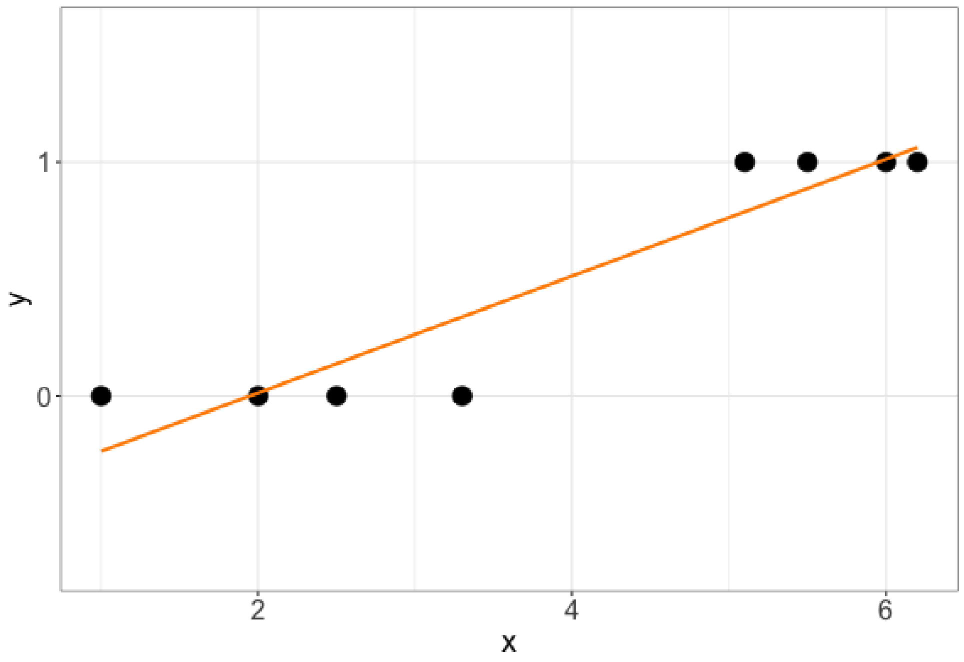
Module completion checklist

Objective	Complete
Determine when to use logistic regression for classification and transformation of target variable	✔
Summarize the process and the math behind logistic regression	
Implement logistic regression on a training dataset and predict on test	
Review classification performance metrics and assess results of logistic model performance	
Transform categorical variables for implementation of logistic regression	
Implement logistic regression on the data and assess results of classification model performance	
Analyze the model to determine if / when overfitting occurs	
Demonstrate tuning the model using grid search cross-validation	

Linear vs logistic regression

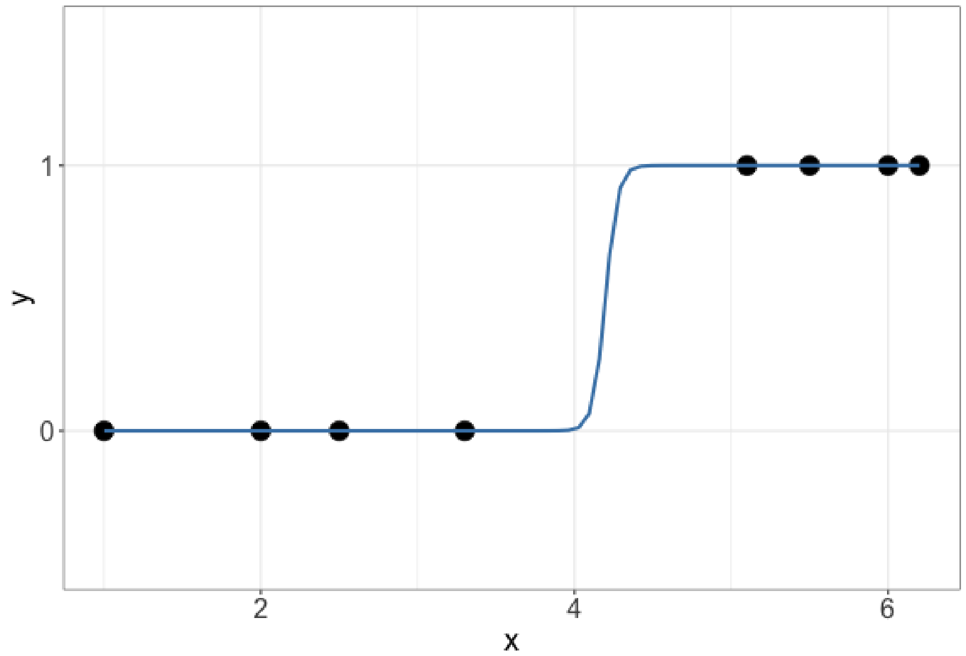
Linear regression line

- For data points $x_1, \ldots, x_n$, we have $y = 0$ or $y = 1$
- The function that "fits" the points is a simple line $\hat{y} = ax + b$



Logistic regression curve

- For the same data points $x_1, \ldots, x_n$, $y = 0$ or $y = 1$
- The function that "fits" the data points is a sigmoid $p(y = 1) = \frac{exp(ax+b)}{1+exp(ax+b)}$



Logistic regression: function

- For every value of $\boldsymbol{x}$, we find $\boldsymbol{p}$, i.e. probability of success, or probability that $\boldsymbol{y} = 1$
- To solve for $\boldsymbol{p}$, logistic regression uses an expression called a **sigmoid function:**

$$p = \frac{exp(ax + b)}{1 + exp(ax + b)}$$

- We can see a very familiar equation inside of the parentheses: $\boldsymbol{ax + b}$

Logistic regression: a bit more math

Through some algebraic transformations that are beyond the scope of this course,

$$p = \frac{exp(ax + b)}{1 + exp(ax + b)}$$

can become

$$logit(p) = log\left(\frac{p}{1 - p}\right)$$

- Since `p` is the **probability of success**, `1 - p` is the **probability of failure**
- The ratio $\left(\frac{p}{1-p}\right)$ is called the **odds** ratio - it tells us the **odds** of having a successful outcome with respect to the opposite
- **Why should we care?**
  - Knowing this provides useful insight into interpreting the coefficients

Logistic regression: coefficients

- In **linear** regression, the coefficients in the equation can easily be interpreted

$$ax + b$$

- An increase in x will result in an increase in y and vice versa

BUT

- In **logistic** regression, the simplest way to interpret a positive coefficient is with an increase in likelihood
- A larger value of x increases the likelihood that $y = 1$

Module completion checklist

Objective	Complete
Determine when to use logistic regression for classification and transformation of target variable	✔
Summarize the process and the math behind logistic regression	✔
Implement logistic regression on a training dataset and predict on test	
Review classification performance metrics and assess results of logistic model performance	
Transform categorical variables for implementation of logistic regression	
Implement logistic regression on the data and assess results of classification model performance	
Analyze the model to determine if / when overfitting occurs	
Demonstrate tuning the model using grid search cross-validation	

Research questions and datasets

- The **credit card dataset** will be used in class to predict which credit card holders will default
- We will start by running a model on a simple subset based on a few predictors (limit balance, demographic data)
- Then, we are going to predict the same with whole dataset

- The **bank marketing dataset** will be used in the exercises to predict if a client has subscribed a term deposit
- Similar to the credit card data, we will first predict using the demographic data (education, marital status, etc..)
- Then, we will include social and economic attributes to predict the subscription of term deposit

Loading data into Python

- Let's load the entire credit card dataset
- We are now going to use the function `read_csv` to read in our `credit_card_data` dataset

```
credit_card = pd.read_csv("credit_card_data.csv")
print(credit_card.head())
```

```
   ID  LIMIT_BAL  SEX  ...  PAY_AMT5  PAY_AMT6  default_payment_next_month
0   1      20000    2  ...         0         0                           1
1   2     120000    2  ...         0      2000                           1
2   3      90000    2  ...      1000      5000                           0
3   4      50000    2  ...      1069      1000                           0
4   5      50000    1  ...       689       679                           0

[5 rows x 25 columns]
```

Renaming target variable

- Rename the target variable to 'default_payment'

```
credit_card = credit_card.rename(columns = {'default_payment_next_month' : 'default_payment'})
print(credit_card.head())
```

```
   ID  LIMIT_BAL  SEX  EDUCATION  ...  PAY_AMT4  PAY_AMT5  PAY_AMT6  default_payment
0   1      20000    2          2  ...         0         0         0                1
1   2     120000    2          2  ...      1000         0      2000                1
2   3      90000    2          2  ...      1000      1000      5000                0
3   4      50000    2          2  ...      1100      1069      1000                0
4   5      50000    1          2  ...      9000       689       679                0

[5 rows x 25 columns]
```

The data at first glance

- Look at the data types of each variable

```
# The data types.
print(credit_card.dtypes)
```

```
ID                    int64
LIMIT_BAL             int64
SEX                   int64
EDUCATION             int64
MARRIAGE              int64
AGE                   int64
PAY_0                 int64
PAY_2                 int64
PAY_3                 int64
PAY_4                 int64
PAY_5                 int64
PAY_6                 int64
BILL_AMT1           float64
BILL_AMT2             int64
BILL_AMT3             int64
BILL_AMT4             int64
BILL_AMT5             int64
BILL_AMT6             int64
PAY_AMT1              int64
PAY_AMT2              int64
PAY_AMT3              int64
PAY_AMT4              int64
PAY_AMT5              int64
PAY_AMT6              int64
default_payment       int64
dtype: object
```

Frequency table of the target variable

- Now let's check the **frequency** of 'default_payment'

```
print(credit_card['default_payment'].value_counts())
```

```
0    23364
1     6636
Name: default_payment, dtype: int64
```

- It has **two levels**, 0 and 1, where **0** is cardholders who **did not** make a default payment

Data prep: check for NAs

- Check for NAs

```
# Check for NAs.
print(credit_card.isnull().sum())
```

- We have 1 missing value in the variable column 'BILL_AMT1'

```
ID                 0
LIMIT_BAL          0
SEX                0
EDUCATION          0
MARRIAGE           0
AGE                0
PAY_0              0
PAY_2              0
PAY_3              0
PAY_4              0
PAY_5              0
PAY_6              0
BILL_AMT1          1
BILL_AMT2          0
BILL_AMT3          0
BILL_AMT4          0
BILL_AMT5          0
BILL_AMT6          0
PAY_AMT1           0
PAY_AMT2           0
PAY_AMT3           0
PAY_AMT4           0
PAY_AMT5           0
PAY_AMT6           0
default_payment    0
dtype: int64
```

Filling missing values

- We will fill the missing value in 'BILL_AMT1' with the mean value

```
# Fill missing values with mean
credit_card = credit_card.fillna(credit_card.mean()['BILL_AMT1'])
```

```
# Check for NAs in 'BILL_AMT1'.
print(credit_card.isnull().sum()['BILL_AMT1'])
```

```
0
```

- Now that we have filled in all NAs, we are ready to scale our **predictors**
- We decided to focus on **limit balance** and **demographic data**

Data prep: numeric variables

- We try and use **numeric data** as predictors
- In some cases, we can **convert categorical data to integer values**
- However, in this simple example, our predictors are numeric by default

- Let's double check:

```
print(credit_card.dtypes.head())
```

```
ID             int64
LIMIT_BAL      int64
SEX            int64
EDUCATION      int64
MARRIAGE       int64
dtype: object
```

Data prep: target

- The next step of our data cleanup is to ensure the target variable is **binary** and has a label
- Let's look at the `dtype` of `default_payment`

```
print(credit_card.default_payment.dtypes)
```

```
int64
```

- We want to convert this to `bool` (Boolean type) so that it's a binary class

```
credit_card["default_payment"] = np.where(credit_card["default_payment"] == 1, True, False)

# Check class again.
print(credit_card.default_payment.dtypes)
```

```
bool
```

Subsetting data

- Now let's **subset** our data so that we have only the variables we need for building our model
- We will drop the variables containing ID as they do not provide any significance for the model
- For our first model, we will only use the demographic data to predict the `default_payment`. So, we will also remove the rest of the predictors
- Let's name this subset `credit_card_glm`

```
credit_card_glm = credit_card[["LIMIT_BAL","SEX","EDUCATION", "MARRIAGE", "AGE", "default_payment"]]
print(credit_card_glm.head())
```

```
   LIMIT_BAL  SEX  EDUCATION  MARRIAGE  AGE  default_payment
0      20000    2          2         1   24             True
1     120000    2          2         2   26             True
2      90000    2          2         2   34            False
3      50000    2          2         1   37            False
4      50000    1          2         1   57            False
```

Split into train and test set

- We're going to split our data into **training** and **test** sets
- We run logistic regression initially on the training data

```
# Separate predictors from data.
X = credit_card_glm[['LIMIT_BAL', 'SEX', 'EDUCATION', 'MARRIAGE', 'AGE']]
```

```
# Separate target from data.
y = np.array(credit_card_glm['default_payment'])
```

```
# Set the seed.
np.random.seed(1)

# Split data into training and test sets, use a 70 train - 30 test split.
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size = .3)
```

scikit-learn - logistic regression

- We will be using the `LogisticRegression` library from `scikit-learn.linear_model` package

`sklearn.linear_model`.LogisticRegression

class `sklearn.linear_model.` **`LogisticRegression`** (*penalty='l2', dual=False, tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None, random_state=None, solver='liblinear', max_iter=100, multi_class='ovr', verbose=0, warm_start=False, n_jobs=1*) [source]

Logistic Regression (aka logit, MaxEnt) classifier.

- All inputs are optional arguments, but we will concentrate on two key inputs:
  - `penalty`: a regularization technique used to tune the model (either `l1`, a.k.a. *Lasso*, or `l2`, a.k.a. *Ridge*, default is `l2`)
  - `C`: a regularization constant used to amplify the effect of the regularization method (a value between $[0, \infty]$ default is `1`)
- For all the parameters of the `LogisticRegression` function, visit ***scikit-learn's documentation***

Logistic regression: solvers and their penalties

We'll be using **liblinear** and **lfbgs** solvers in this module, but there are others

Solver	Behavior	Penalty
liblinear	Ideal for small datasets and one vs rest schemes	L1 and L2
lbfgs	Default solver, ideal for large data sets and multi-class problems	L2 or no penalty
newton-cg	Ideal for large data sets and multi-class problems	L2 or no penalty
sag	Works faster on large data sets and handles multi-class problems	L2 or no penalty
saga	Works faster on large data sets and handles multi-class problems	L1, L2, elastic net or no penalty

- To learn more about solvers in logistic regression, visit ***scikit-learn's documentation***

Logistic regression: build

- Let's build our logistic regression model
- We'll use all default parameters for now as our baseline model

```
# Set up logistic regression model.
logistic_regression_model = linear_model.LogisticRegression()
print(logistic_regression_model)
```

```
LogisticRegression()
```

- We can see that the default model contains `C = 1` and `penalty = 'l2'`
- We will discuss what that means when we tune our model later

Logistic regression: fit

The two main arguments are the same as with most classifiers in `scikit-learn`:

1. `X`: a `pandas` dataframe or a `numpy` array of training data predictors
2. `y`: a `pandas` series or a `numpy` array of training labels

`fit` (*X*, *y*, *sample_weight=None*) [source]

Fit the model according to the given training data.

Parameters:	**X** : {array-like, sparse matrix}, shape (n_samples, n_features) Training vector, where n_samples is the number of samples and n_features is the number of features. **y** : array-like, shape (n_samples,) Target vector relative to X. **sample_weight** : array-like, shape (n_samples,) optional Array of weights that are assigned to individual samples. If not provided, then each sample is given unit weight. *New in version 0.17: sample_weight* support to LogisticRegression.
Returns:	**self** : object Returns self.

Logistic regression: fit

- We fit the logistic regression model with `X_train` and `y_train`
- We will run the model on our training data and predict on test data

```
# Fit the model.
logistic_regression_model.fit(X_train,
                              y_train)
```

```
LogisticRegression()
```

Logistic regression: predict

- The main argument is the same as with most classifiers in `scikit-learn`:
  - `X`: a `pandas` dataframe or a `numpy` array of test data predictors

predict (*X*) [source]

Predict class labels for samples in X.

Parameters:	**X** : {array-like, sparse matrix}, shape = [n_samples, n_features] Samples.
Returns:	**C** : array, shape = [n_samples] Predicted class label per sample.

Logistic regression: predict

- We will predict on the test data using our trained model
- The result is a **vector of the predictions**
- The instances yielding `True` are predicted to have made the `default_payment`

```
# Predict on test data.
predicted_values = logistic_regression_model.predict(X_test)
print(predicted_values)
```

```
[False False False ... False False False]
```

Knowledge check 1

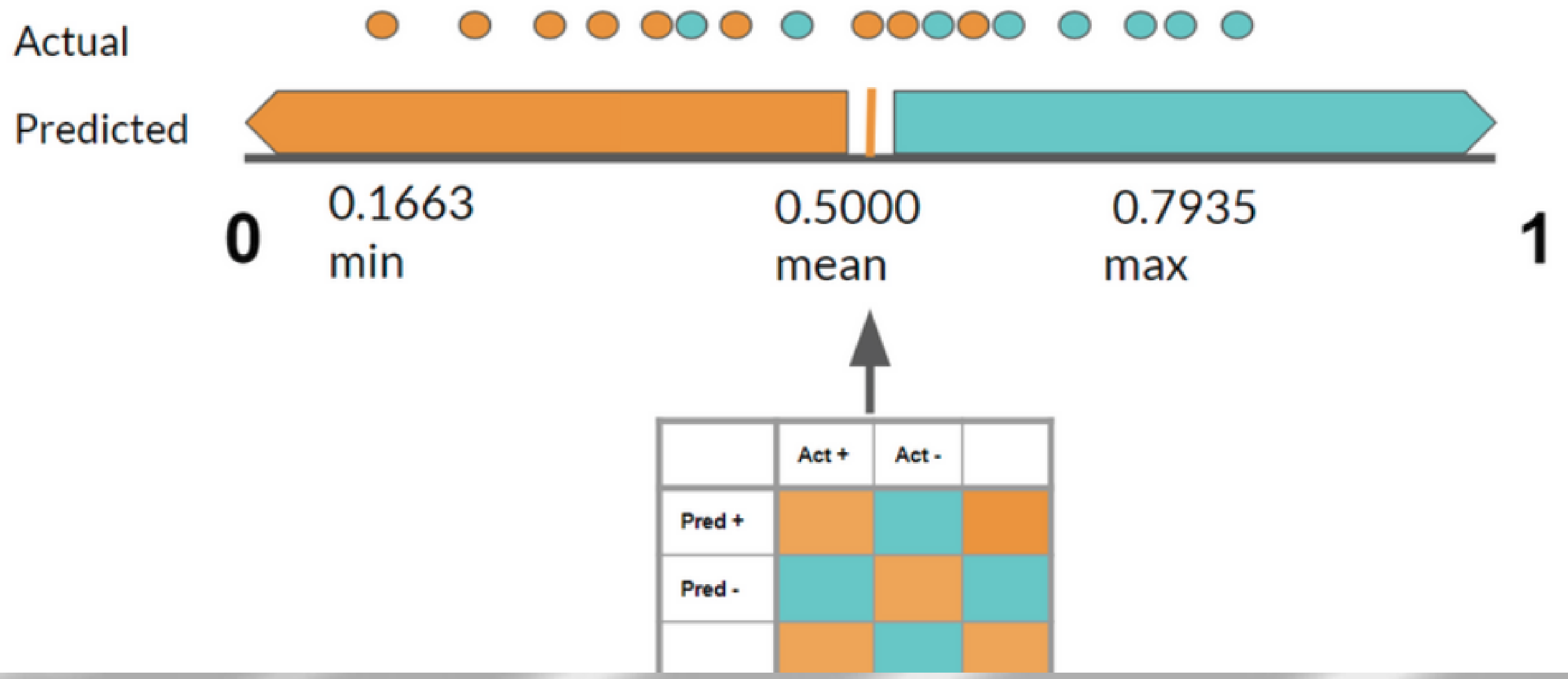
Exercise 1

Module completion checklist

Objective	Complete
Determine when to use logistic regression for classification and transformation of target variable	✔
Summarize the process and the math behind logistic regression	✔
Implement logistic regression on a training dataset and predict on test	✔
Review classification performance metrics and assess results of logistic model performance	
Transform categorical variables for implementation of logistic regression	
Implement logistic regression on the data and assess results of classification model performance	
Analyze the model to determine if / when overfitting occurs	
Demonstrate tuning the model using grid search cross-validation	

From threshold to metrics

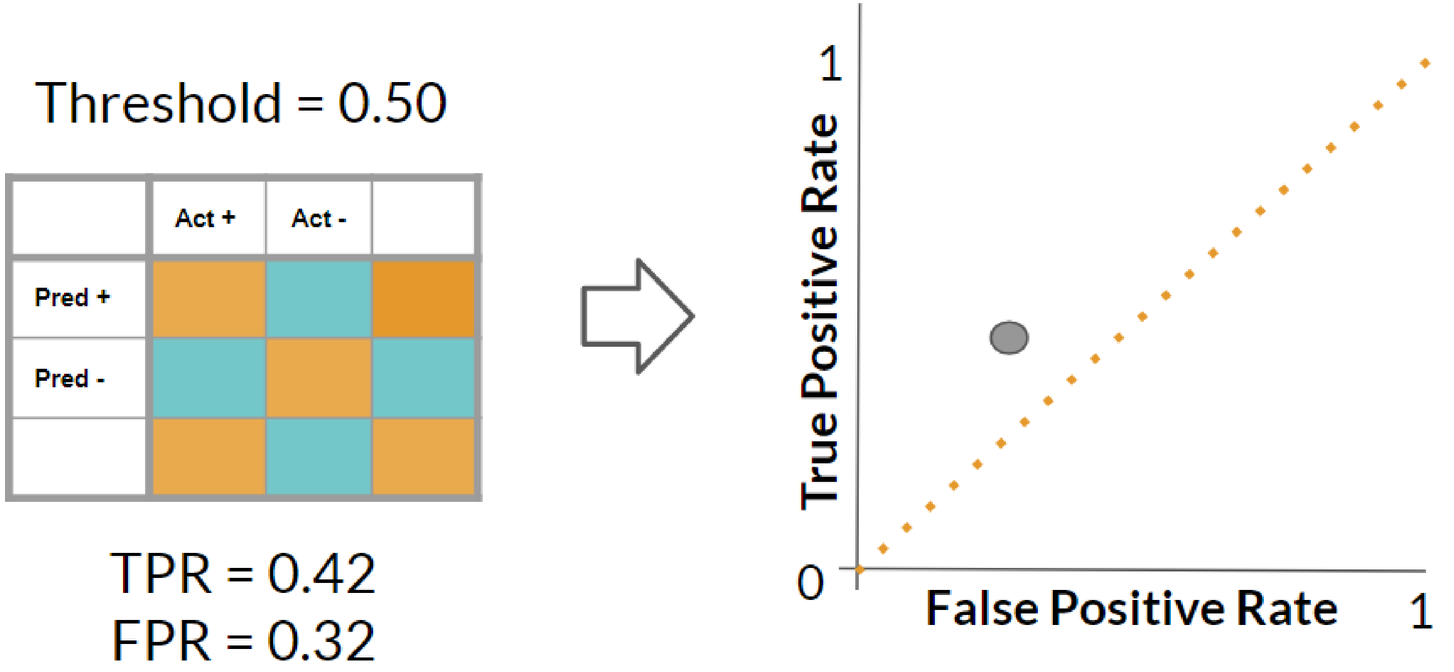
- In logistic regression, the output is a range of probabilities from `0` to `1`
- But how do you interpret that as a `1` / `0` or `High value` / `Low value` label?
- You set a **threshold** where everything above is predicted as `1` and everything below is predicted as `0`
- A typical threshold for logistic regression is `0.5` but can be any value in the range of `0` to `1`



From metrics to a point

Each threshold can create a **confusion matrix**, which can be used to calculate a point in space defined by:

- **True positive rate (TPR)** on the `y-axis`
- **False positive rate (FPR)** on the `x-axis`

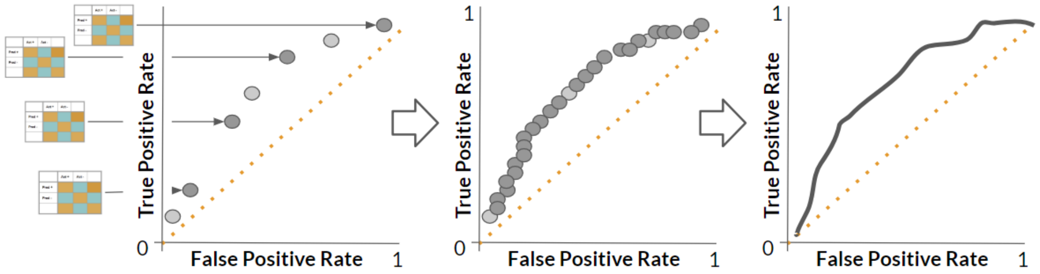


Confusion matrix

	Predicted Low value	Predicted High value	Actual totals
Actual low value	**True negative (TN)**	**False positive (FP)**	Total negatives
Actual high value	**False negative (FN)**	**True positive (TP)**	Total positives
Predicted totals	Total predicted negatives	Total predicted positives	**Total**

- **True positive rate (TPR)** (a.k.a. *Sensitivity*, *Recall*) = **TP** / Total positives
- **True negative rate (TNR)** (a.k.a. *Specificity*) = **TN** / Total negatives
- **False positive rate (FPR)** (a.k.a. *Fall-out*, *Type I Error*) = **FP** / Total negatives
- **False negative rate (FNR)** (a.k.a. *Type II Error*) = **FN** / Total positives
- **Accuracy** = **TP + TN** / **Total**
- **Misclassification rate** = **FP + FN** / **Total**

From points to a curve



- When we move thresholds, we re-calculate our metrics and create confusion matrices for every threshold
- Each time, we plot a new point in the **TPR** vs **FPR** space

ROC curve

- The **receiver operating characteristic curve** is a performance metric used to compare classification models to measure predictive accuracy
- The **AUC (area under curve)** should be above .5 to say the model is better than a random guess
- As the AUC approaches 1, the more likely the model is to maximize TPR and minimize FPR.

scikit-learn: metrics package

sklearn.metrics: Metrics

See the Model evaluation: quantifying the quality of predictions section and the Pairwise metrics, Affinities and Kernels section of the user guide for further details.

The `sklearn.metrics` module includes score functions, performance metrics and pairwise metrics and distance computations.

- We will use the following methods from this library:
  - `confusion_matrix`
  - `accuracy_score`
  - `classification_report`
  - `roc_curve`
  - `auc`
- For all the methods and parameters of the `metrics` package, visit ***scikit-learn's documentation***

Confusion matrix and accuracy

Both `confusion_matrix` and `accuracy_score` take **two** arguments:

1. Original data labels
2. Predicted labels

```
# Take a look at test data confusion matrix.
conf_matrix_test = metrics.confusion_matrix(y_test, predicted_values)
print(conf_matrix_test)
```

```
[[7000    0]
 [2000    0]]
```

```
# Compute test model accuracy score.
test_accuracy_score = metrics.accuracy_score(y_test, predicted_values)
print("Accuracy on test data: ", test_accuracy_score)
```

```
Accuracy on test data:  0.7777777777777778
```

Classification report

- To make interpretation of the `classification_report` easier, in addition to the two arguments that `confusion_matrix` takes, we can add the actual class names for our target variable

```
# Create a list of target names to interpret class assignments.
target_names = ['default_payment_0', 'default_payment_1']
```

```
# Print an entire classification report.
class_report = metrics.classification_report(y_test,
                                              predicted_values,
                                              target_names = target_names)
print(class_report)
```

```
                   precision    recall  f1-score   support

default_payment_0       0.78      1.00      0.88      7000
default_payment_1       0.00      0.00      0.00      2000

         accuracy                           0.78      9000
        macro avg       0.39      0.50      0.44      9000
     weighted avg       0.60      0.78      0.68      9000
```

Precision

	Positive	Negative
Positive	TP	**FP**
Negative	**FN**	TN

- $PR = \frac{(TP)}{(TP+FP)}$
- A proportion of values that is truly positive out of all predicted positive values
- a.k.a. PPV - positive predicted value
- What percent of your predictions were correct?

Recall

	Positive	Negative
Positive	TP	**FP**
Negative	**FN**	TN

- $RE = \frac{(TP)}{(TP+FN)}$
- Proportion of actual positives that is classified correctly
- a.k.a. sensitivity, hit rate, or true positive rate (TPR)
- What percent of the positive cases did you catch?

F1: precision vs recall

- A score that gives us a numeric value of the precision vs recall tradeoff
- What percent of positive predictions were correct?
- `f1-score` is calculated as a weighted harmonic mean of `precision` and `recall`
- $F1 = 2 \times \frac{(PR*RE)}{(PR+RE)}$
- The higher the $F1$ score, the better (the score can be a value between 0 and 1)
- **Support** is the actual number of occurrences of each class in `y_test`

Pickle - what?

- Now that we have explored this model's data, we can use a function in Python called `pickle` to save it for later
- We `pickle` objects we want to save from one script/session to pull up in new scripts, without rerunning code
- It is similar to **flattening** a file
  - **Pickle/saving: a Python object is converted into a byte stream**
  - **Unpickle/loading: the inverse operation where a byte stream is converted back into an object**

Model champion dataframe

- Let's create a dataframe, store the accuracy and then pickle the dataframe
- This way, we can use the `model_final` dataframe across all our classification algorithms to choose our final model champion!

```
# Create a dictionary with accuracy values for our logistic regression model.
model_final_dict = {'metrics': ["accuracy"],
                    'values':[round(test_accuracy_score, 4)],
                    'model':['logistic']}

model_final = pd.DataFrame(data = model_final_dict)
print(model_final)
```

```
    metrics  values     model
0  accuracy  0.7778  logistic
```

```
pickle.dump(model_final, open("model_final.sav","wb" ))
```

Getting probabilities instead of class labels

```
# Get probabilities instead of predicted values.
test_probabilities = logistic_regression_model.predict_proba(X_test)
print(test_probabilities[0:5, :])
```

```
[[0.89322323 0.10677677]
 [0.51712316 0.48287684]
 [0.58482104 0.41517896]
 [0.6337072  0.3662928 ]
 [0.86412936 0.13587064]]
```

```
# Get probabilities of test predictions only.
test_predictions = test_probabilities[:, 1]
print(test_predictions[0:5])
```

```
[0.10677677 0.48287684 0.41517896 0.3662928  0.13587064]
```

Computing FPR, TPR, and threshold

```
# Get FPR, TPR, and threshold values.
fpr, tpr, threshold = metrics.roc_curve(y_test,          #<- test data labels
                                        test_predictions)  #<- predicted probabilities
print("False positive: ", fpr[:5])
```

```
False positive:  [0.         0.00028571 0.00042857 0.00057143 0.00071429]
```

```
print("True positive: ", tpr[:5])
```

```
True positive:  [0.     0.0005 0.001  0.001  0.0015]
```

```
print("Threshold: ", threshold[:5])
```

```
Threshold:  [1.48287687 0.48287687 0.48287687 0.48287687 0.48287687]
```

Computing AUC

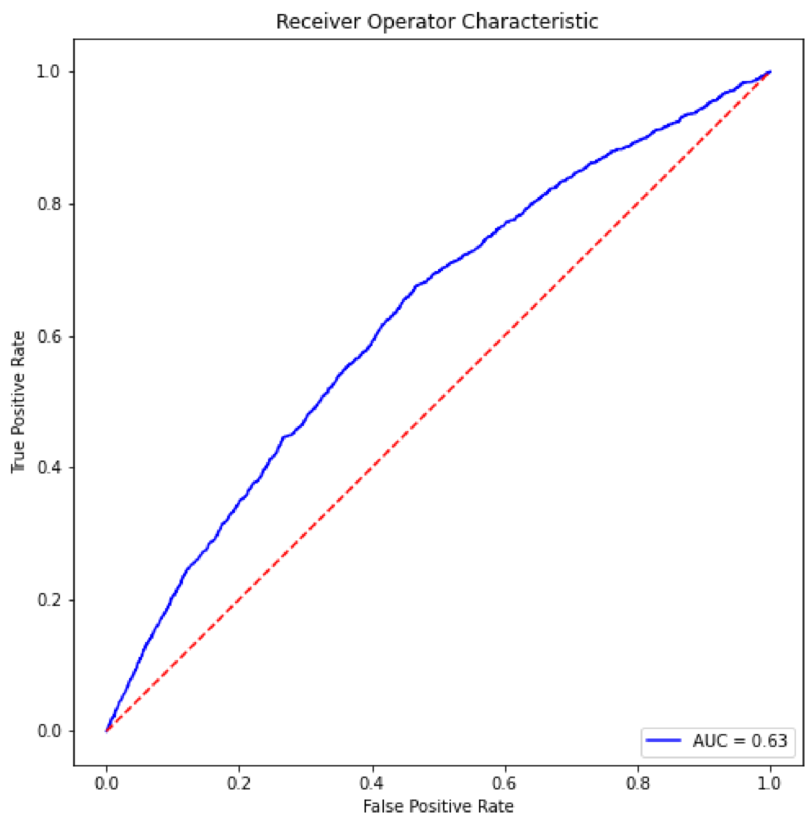
```
# Get AUC by providing the FPR and TPR.
auc = metrics.auc(fpr, tpr)
print("Area under the ROC curve: ", auc)
```

```
Area under the ROC curve:  0.6280798214285714
```

Putting it all together: ROC plot

```
# Make an ROC curve plot.
plt.title('Receiver Operator Characteristic')
plt.plot(fpr, tpr, 'b', label = 'AUC = %0.2f' % auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
```

- Our model achieved the accuracy of about `0.78`, which is decent for a base model.
- Our estimated AUC is about `0.63`
- Given that we have not done any model tuning or data transformations, this is a fair baseline that we'll use to assess future models that we'll create



Knowledge check 2

Exercise 2

Module completion checklist

Objective	Complete
Determine when to use logistic regression for classification and transformation of target variable	✔
Summarize the process and the math behind logistic regression	✔
Implement logistic regression on a training dataset and predict on test	✔
Review classification performance metrics and assess results of logistic model performance	✔
Transform categorical variables for implementation of logistic regression	
Implement logistic regression on the data and assess results of classification model performance	
Analyze the model to determine if / when overfitting occurs	
Demonstrate tuning the model using grid search cross-validation	

Working with categorical variables

- Let's take a look at numerical variable `age` from our dataset

```
print(credit_card.AGE.head())
```

```
0    24
1    26
2    34
3    37
4    57
Name: AGE, dtype: int64
```

- We are going to convert `age` to a **categorical variable with 3 levels**, to analyze varying subscription to term deposit between age groups

```
credit_card['AGE'] = np.where(credit_card['AGE'] <= 30, "30 or Below",
             np.where(credit_card['AGE'] < 60, 'Between 30 and 60', '60 and above'))
```

Working with categorical variables

- Let's see the frequency of each level in `age`

```
credit_card.AGE.value_counts()
```

```
Between 30 and 60     18648
30 or Below           11013
60 and above            339
Name: AGE, dtype: int64
```

- As regression analysis is used with **numeric or continuous variables** to determine an outcome, how would we handle **categorical variables**?

One-hot encoding

- It creates an **artificial variable** used to represent a variable with **two or more distinct levels or categories**
- It represents categorical predictors as binary values, **0 or 1**
- Often used for **regression analysis**

ID	Pet
1	Dog
2	Cat
3	Cat
4	Dog
5	Fish

ID	Dog	Cat	Fish
1	1	0	0
2	0	1	0
3	0	1	0
4	1	0	0
5	0	0	1

Reference category

- The number of dummy variables necessary to represent a single attribute variable is equal to the **number of levels (categories) in that variable minus one**
- One of the categories is omitted and used as a **base or reference category**
- The reference category, which is not coded, is the category to which **all other categories will be compared**
- The biggest group / category will often be the reference category

Dummy variables in Python

```
pd.get_dummies(dataframe['Column'],
               drop_first = ,
               ...)
```

- `data` is a pandas `Series` or `Dataframe`
- `drop_first` indicates whether to get `k-1` dummies out of `k` categorical levels

pandas.get_dummies

`pandas.get_dummies`(*data, prefix=None, prefix_sep='_', dummy_na=False, columns=None, sparse=False, drop_first=False, dtype=None*) [source]

Convert categorical variable into dummy/indicator variables

Parameters:

data : *array-like, Series, or DataFrame*

prefix : *string, list of strings, or dict of strings, default None*
String to append DataFrame column names. Pass a list with length equal to the number of columns when calling get_dummies on a DataFrame. Alternatively, *prefix* can be a dictionary mapping column names to prefixes.

prefix_sep : *string, default '_'*
If appending prefix, separator/delimiter to use. Or pass a list or dictionary as with *prefix*.

dummy_na : *bool, default False*
Add a column to indicate NaNs, if False NaNs are ignored.

columns : *list-like, default None*
Column names in the DataFrame to be encoded. If *columns* is None then all the columns with *object* or *category* dtype will be converted.

sparse : *bool, default False*
Whether the dummy-encoded columns should be be backed by a `SparseArray` (True) or a regular NumPy array (False).

drop_first : *bool, default False*
Whether to get k-1 dummies out of k categorical levels by removing the first level.
New in version 0.18.0.

dtype : *dtype, default np.uint8*
Data type for new columns. Only a single dtype is allowed.
New in version 0.23.0.

Returns:

dummies : *DataFrame*

Transform age variable into a dummy variable

- We need to transform `age`, which is categorical variable with 3 levels, into a dummy variable and save it into a dataframe

```
# Convert 'age' into dummy variables.
age_dummy = pd.get_dummies(credit_card['AGE'], drop_first = True)
print(age_dummy.head())
```

```
   60 and above  Between 30 and 60
0             0                  0
1             0                  0
2             0                  1
3             0                  1
4             0                  1
```

- Notice that level `30 or below`, which has the highest count, has been removed and used as a reference category

Drop age and replace with the dummy variable

- Let's drop the original `age` column from our credit card subset and concatenate the dummy variables `age_dummy`

```
# Drop `age` from the data.
credit_card.drop(['AGE'], axis = 1, inplace = True)
```

```
# Concatenate `age_dummy` to our dataset.
credit_card = pd.concat([credit_card,age_dummy],axis=1)
print(credit_card.head())
```

```
   ID  LIMIT_BAL  SEX  ...  default_payment  60 and above  Between 30 and 60
0   1      20000    2  ...             True             0                  0
1   2     120000    2  ...             True             0                  0
2   3      90000    2  ...            False             0                  1
3   4      50000    2  ...            False             0                  1
4   5      50000    1  ...            False             0                  1

[5 rows x 26 columns]
```

Transform and replace other categorical variables

- Let's transform the remaining categorical values into dummy variables and save it into a dataframe

```
# Convert 'sex' into dummy variables.
sex_dummy = pd.get_dummies(credit_card['SEX'], prefix = 'sex', drop_first = True)
# Convert 'education' into dummy variables.
education_dummy = pd.get_dummies(credit_card['EDUCATION'], prefix = 'education', drop_first = True)
# Convert 'marriage' into dummy variables.
marriage_dummy = pd.get_dummies(credit_card['MARRIAGE'], prefix = 'marriage', drop_first = True)
```

```
# Drop `sex`, `education`, `marriage` from the data.
credit_card.drop(['SEX', 'EDUCATION', 'MARRIAGE'], axis = 1, inplace = True)
```

```
# Concatenate `sex_dummy`, `education_dummy`, `marriage_dummy` to our dataset.
credit_card = pd.concat([credit_card,sex_dummy,education_dummy,marriage_dummy],axis=1)
print(credit_card.head())
```

```
   ID  LIMIT_BAL  PAY_0  PAY_2  ...  education_6  marriage_1  marriage_2  marriage_3
0   1      20000      2      2  ...            0           1           0           0
1   2     120000     -1      2  ...            0           0           1           0
2   3      90000      0      0  ...            0           0           1           0
3   4      50000      0      0  ...            0           1           0           0
4   5      50000     -1      0  ...            0           1           0           0

[5 rows x 33 columns]
```

Module completion checklist

Objective	Complete
Determine when to use logistic regression for classification and transformation of target variable	✔
Summarize the process and the math behind logistic regression	✔
Implement logistic regression on a training dataset and predict on test	✔
Review classification performance metrics and assess results of logistic model performance	✔
Transform categorical variables for implementation of logistic regression	✔
Implement logistic regression on the data and assess results of classification model performance	
Analyze the model to determine if / when overfitting occurs	
Demonstrate tuning the model using grid search cross-validation	

Split into train and test set

- Let's train the logistic regression model initially on the training data and then test its performance on the test data
- We will need to split our data into 2 parts, where 70% will go to the training set, and the remaining 30% will be used for our model test

```
# Separate predictors from data.
# We can just drop the target variable, as we are using all other variables as predictors.
X = credit_card.drop('default_payment', axis = 1)
```

```
# Separate target from data.
y = np.array(credit_card['default_payment'])
```

```
# Set the seed.
np.random.seed(1)
# Split data into training and test sets, use a 70 train - 30 test split.
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size = .3)
```

Logistic regression: build

sklearn.linear_model.LogisticRegression

class sklearn.linear_model. **LogisticRegression** (*penalty='l2', dual=False, tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None, random_state=None, solver='liblinear', max_iter=100, multi_class='ovr', verbose=0, warm_start=False, n_jobs=1*) [source]

Logistic Regression (aka logit, MaxEnt) classifier.

```
# Set up the logistic regression model.
logistic_regression_model = linear_model.LogisticRegression(solver='liblinear')
print(logistic_regression_model)
```

```
LogisticRegression(solver='liblinear')
```

- `LogisticRegression` function is supplied with different solvers, penalty parameters and other tuning tools, in this iteration we will use `liblinear` solver, which uses a coordinate descent (CD) algorithm and is well suited for one-vs-all schemes
- Take a look at `Logistic regression: solvers and their penalties` (slide 38)
- To learn more about function parameters, visit ***scikit-learn's documentation***

Logistic regression: fit

- We fit the logistic regression model with `X_train` and `y_train`
- We will run the model on our training data and predict on test data

```
# Fit the model.
logistic_regression_model.fit(X_train,
                              y_train)
```

```
LogisticRegression(solver='liblinear')
```

Logistic regression: predict

- We will predict on the test data using our trained model
- The result is a **vector of the predictions**

```
# Predict on test data.
predicted_values = logistic_regression_model.predict(X_test)
print(predicted_values)
```

```
[False False False ... False False False]
```

Confusion matrix and accuracy

Both `confusion_matrix` and `accuracy_score` take two arguments:

1. Original data labels
2. Predicted labels

```
# Take a look at test data confusion matrix.
conf_matrix_test = metrics.confusion_matrix(y_test, predicted_values)
print(conf_matrix_test)
```

```
[[7000    0]
 [2000    0]]
```

```
# Compute test model accuracy score.
test_accuracy_score = metrics.accuracy_score(y_test, predicted_values)
print("Accuracy on test data: ", test_accuracy_score)
```

```
Accuracy on test data:  0.7777777777777778
```

Add accuracy score to the final scores

- So we have it, let's add this score to the dataframe `model_final` that we created previously
- Let's load the pickled dataset and append the score to it

```
model_final = pickle.load(open("model_final.sav","rb"))
```

```
model_final = model_final.append({'metrics' : "accuracy" ,
                                  'values' : round(test_accuracy_score,4),
                                  'model':'logistic_whole_dataset'} ,
                                  ignore_index = True)
print(model_final)
```

```
    metrics  values                   model
0  accuracy  0.7778                logistic
1  accuracy  0.7778  logistic_whole_dataset
```

Accuracy on train vs accuracy on test

- Take a look at the accuracy score for the training data

```
# Compute trained model accuracy score.
trained_accuracy_score = logistic_regression_model.score(X_train, y_train)
print("Accuracy on train data: " , trained_accuracy_score)
```

```
Accuracy on train data:  0.7792380952380953
```

- Did our model underperform?
- Is there a big difference in `train` and `test` accuracy?
- Most of the time, the problem lies in **overfitting**

Knowledge check 3

Exercise 3

Module completion checklist

Objective	Complete
Determine when to use logistic regression for classification and transformation of target variable	✔
Summarize the process and the math behind logistic regression	✔
Implement logistic regression on a training dataset and predict on test	✔
Review classification performance metrics and assess results of logistic model performance	✔
Transform categorical variables for implementation of logistic regression	✔
Implement logistic regression on the data and assess results of classification model performance	✔
Analyze the model to determine if / when overfitting occurs	
Demonstrate tuning the model using grid search cross-validation	

When overfitting occurs

- An overfitted model usually shows a drastically higher accuracy in the training data because it **doesn't generalize well to new data**
- Creating a model that fits training data **too well** will lead to poor generalization and, hence, poor performance on new data. It can happen for a number of reasons:
  - the model treats the **noise** as actual artifacts of the data, so when it encounters new data with new **noise**, the model will underperform
  - by using **too many predictors** that only contribute tiny portions to variation in our data, there is a higher likelihood of overfitting
  - if the training set is **not an accurate representation of the data**, we end up fitting the model to just a part of it, which doesn't translate well to new data

How to overcome overfitting

- Use so-called **soft-margin** classifiers to:
  - Make the model less prone to noise through penalization constants and other methods
  - Tune them to use the optimal parameters for best model performance
- Use **feature selection**, and/or **feature extraction** methods to:
  - Capture only few main features responsible for most variation in the data
  - Discard those that aren't as responsible
- **Get more data**

Tuning logistic regression model

- Recall the two parameters that we mentioned before:
  - `penalty`: a regularization technique used to tune the model (either `l1`, a.k.a. *Lasso*, or `l2`, a.k.a. *Ridge*; default is `l2`)
  - `C`: a regularization constant used to amplify the effect of the regularization method (a value between $[0, \infty]$; default is `1`)
- These two parameters control a so-called **regularization term** that adds a penalty as the model complexity increases with added variables
- They play a key role in **mitigating overfitting** and **feature pruning**

Regularization techniques in logistic regression

- As you may know, any ML algorithm optimizes some *cost function* $f(x)$
- In logistic regression, `l1` (*Lasso*) adds a term to that function like so:

$$f(x) + C\sum_{j=1}^{n} |b_j|$$

- While `l2` (*Ridge*) adds a term like so:

$$f(x) + C\sum_{j=1}^{n} b_j^2$$

- You can see that *Lasso* uses the absolute value b_j, while *Ridge* uses a squared b_j
- That term, when added to the original *cost function*, **dampens** the margins of our classifier, making it more **forgiving** of the misclassification of some points that might be noise

Lasso vs Ridge

Lasso (l1)

$$C \sum_{j=1}^{n} |b_j|$$

- Stands for **L**east **A**bsolute **S**hrinkage and **S**election **O**perator
- It adds "absolute value of magnitude" of the coefficient as a penalty term to the loss function
- Shrinks (as the name suggests) the less important features' coefficients to zero, which leads to **removal** of some features

Ridge (l2)

$$C \sum_{j=1}^{n} b_j^2$$

- Adds "squared magnitude" of coefficient as penalty term to the loss function
- Dampens the less important features' coefficients making them less significant, which leads to **weighting** of the features according to their importance

What is the role of C?

There are 4 scenarios that might happen with a classifier with respect to $\boldsymbol{C}$:

1. $C = 0$
   - The classifier becomes an **OLS** problem (i.e. Ordinary Least Squares, or just a strict regression without any penalization)
   - Since $0 \times anything = 0$, we are just left with optimizing $f(x)$, which is a definite **overfitting** problem
2. $C = small$
   - We still run into an **overfitting** problem
   - Since $\boldsymbol{C}$ will not "magnify" the effect of the penalty term enough

What is the role of C?

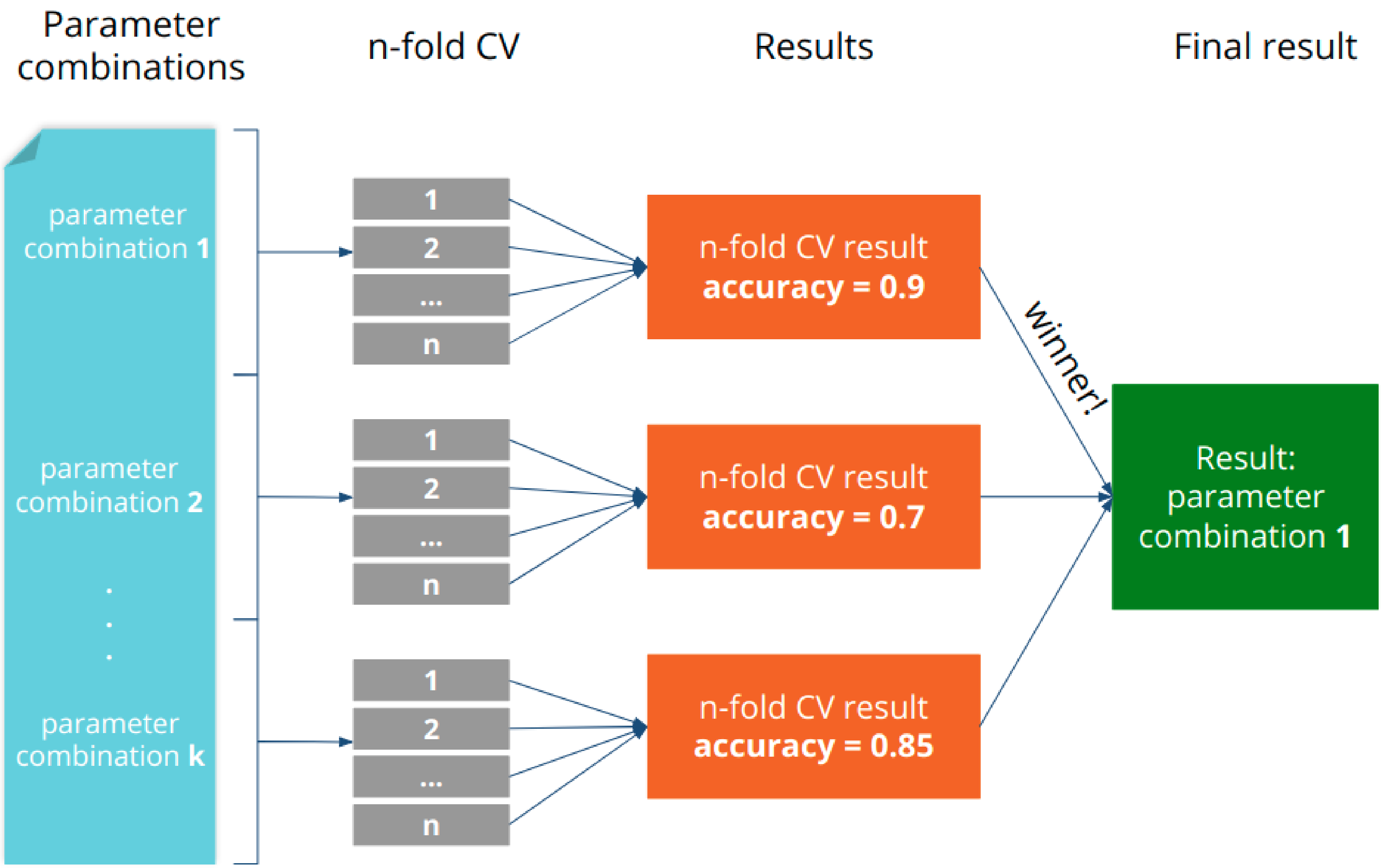
1. $C = large$
   - We run into an **underfitting** problem, where we've weighted and dampened the coefficients too much and we made the model too general
2. $C = optimal$
   - We have a **good, robust, and generalizable model** that works well with new data
   - Ignores most of the noise while preserving the main pattern in data

So how do we pick the right combination of parameters? We use **grid search cross-validation** to find the optimal parameters for our model!

Module completion checklist

Objective	Complete
Determine when to use logistic regression for classification and transformation of target variable	✔
Summarize the process and the math behind logistic regression	✔
Implement logistic regression on a training dataset and predict on test	✔
Review classification performance metrics and assess results of logistic model performance	✔
Transform categorical variables for implementation of logistic regression	✔
Implement logistic regression on the data and assess results of classification model performance	✔
Analyze the model to determine if / when overfitting occurs	✔
Demonstrate tuning the model using grid search cross-validation	

What does grid search cross-validation do?



scikit-learn - model_selection.GridSearchCV

`sklearn.model_selection`.GridSearchCV

class `sklearn.model_selection.` **`GridSearchCV`** (*estimator*, *param_grid*, *scoring=None*, *fit_params=None*, *n_jobs=1*, *iid=True*, *refit=True*, *cv=None*, *verbose=0*, *pre_dispatch='2*n_jobs'*, *error_score='raise'*, *return_train_score='warn'*) [source]

Exhaustive search over specified parameter values for an estimator.

Important members are fit, predict.

GridSearchCV implements a "fit" and a "score" method. It also implements "predict", "predict_proba", "decision_function", "transform" and "inverse_transform" if they are implemented in the estimator used.

The parameters of the estimator used to apply these methods are optimized by cross-validated grid-search over a parameter grid.

- `estimator` is the name of `sklearn` algorithm to optimize
- `param_grid` is a dictionary or list of parameters to optimize
- `cv` is an `int` of `n` for `n-fold` cross-validation
- `verbose` is an `int` of how much verbosity in messages you want to see as the function runs

For all the methods and parameters of the `model_selection.GridSearchCV` package, visit ***scikit-learn's documentation***

Prepare parameters for optimization

```
# Create regularization penalty space.
penalty = ['l1', 'l2']
```

```
# Create regularization constant space.
C = np.logspace(0, 10, 10)
print("Regularization constant: ", C)
```

```
Regularization constant:  [1.00000000e+00 1.29154967e+01 1.66810054e+02 2.15443469e+03
 2.78255940e+04 3.59381366e+05 4.64158883e+06 5.99484250e+07
 7.74263683e+08 1.00000000e+10]
```

```
# Create hyperparameter options dictionary.
hyperparameters = dict(C = C, penalty = penalty)
print(hyperparameters)
```

```
{'C': array([1.00000000e+00, 1.29154967e+01, 1.66810054e+02, 2.15443469e+03,
       2.78255940e+04, 3.59381366e+05, 4.64158883e+06, 5.99484250e+07,
       7.74263683e+08, 1.00000000e+10]), 'penalty': ['l1', 'l2']}
```

Set up cross-validation logistic function

```
# Grid search 10-fold cross-validation with above parameters.
clf = GridSearchCV(linear_model.LogisticRegression(solver='liblinear'), #<- function to optimize
                   hyperparameters,                                     #<- grid search parameters
                   cv = 10,                                             #<- 10-fold cv
                   verbose = 0)                                         #<- no messages to show
```

```
# Fit CV grid search.
best_model = clf.fit(X_train, y_train)
best_model
```

```
GridSearchCV(cv=10, error_score='raise-deprecating',
       estimator=LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='liblinear',
          tol=0.0001, verbose=0, warm_start=False),
       fit_params=None, iid='warn', n_jobs=None,
       param_grid={'C': array([1.00000e+00, 1.29155e+01, 1.66810e+02, 2.15443e+03, 2.78256e+04,
       3.59381e+05, 4.64159e+06, 5.99484e+07, 7.74264e+08, 1.00000e+10]), 'penalty': ['l1', 'l2']},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring=None, verbose=0)
```

Check best parameters found by CV

```
# Get best penalty and constant parameters.
penalty = best_model.best_estimator_.get_params()['penalty']
constant = best_model.best_estimator_.get_params()['C']
print('Best penalty: ', penalty)
```

```
Best penalty:  l1
```

```
print('Best C: ', constant)
```

```
Best C:  12.91549665014884
```

- It seems like our grid search CV has found that `l1` (i.e. *Lasso* regularization method) works better than the default `l2` (i.e. *Ridge*)
- It also shows that the default `C`, which is `1`, creates a big enough soft margin for our classifier

Predict using the best model parameters

```
# Predict on test data using best model.
best_predicted_values = best_model.predict(X_test)
print(best_predicted_values)
```

```
[False False False ... False False False]
```

```
# Compute best model accuracy score.
best_accuracy_score = metrics.accuracy_score(y_test, best_predicted_values)
print("Accuracy on test data (best model): ", best_accuracy_score)
```

```
Accuracy on test data (best model):  0.8076666666666666
```

Predict using the best model parameters (cont'd)

```
# Compute confusion matrix for best model.
best_confusion_matrix = metrics.confusion_matrix(y_test, best_predicted_values)
print(best_confusion_matrix)
```

```
[[6814  186]
 [1545  455]]
```

```
# Create a list of target names to interpret class assignments.
target_names = ['default_payment_no', 'default_payment_yes']
```

```
# Compute classification report for best model.
best_class_report = metrics.classification_report(y_test, best_predicted_values,
                                                  target_names = target_names)
print(best_class_report)
```

```
              precision    recall  f1-score   support

          no       0.82      0.97      0.89      7000
         yes       0.71      0.23      0.34      2000

   micro avg       0.81      0.81      0.81      9000
   macro avg       0.76      0.60      0.62      9000
weighted avg       0.79      0.81      0.77      9000
```

Add accuracy score to the final scores

- Let's add this score to the dataframe `model_final` that we created previously
- We have already loaded the pickled dataframe, so no need to load it again
- Let's append the score to it and dump again for future use

```
model_final = model_final.append({'metrics' : "accuracy",
                                  'values' : round(best_accuracy_score, 4),
                                  'model':'logistic_tuned' } ,
                                  ignore_index = True)
print(model_final)
```

```
    metrics  values                  model
0  accuracy  0.7778               logistic
1  accuracy  0.7778  logistic_whole_dataset
2  accuracy  0.8077         logistic_tuned
```

```
pickle.dump(model_final, open("model_final.sav","wb" ))
```

Get metrics for ROC curve

```
# Get probabilities instead of predicted values.
best_test_probabilities = best_model.predict_proba(X_test)
print(best_test_probabilities[0:5, ])
```

```
[[0.85118308 0.14881692]
 [0.86161466 0.13838534]
 [0.90320762 0.09679238]
 [0.37354264 0.62645736]
 [0.59141147 0.40858853]]
```

```
# Get probabilities of test predictions only.
best_test_predictions = best_test_probabilities[:, 1]
print(best_test_predictions[0:5])
```

```
[0.14881692 0.13838534 0.09679238 0.62645736 0.40858853]
```

Get metrics for ROC curve (cont'd)

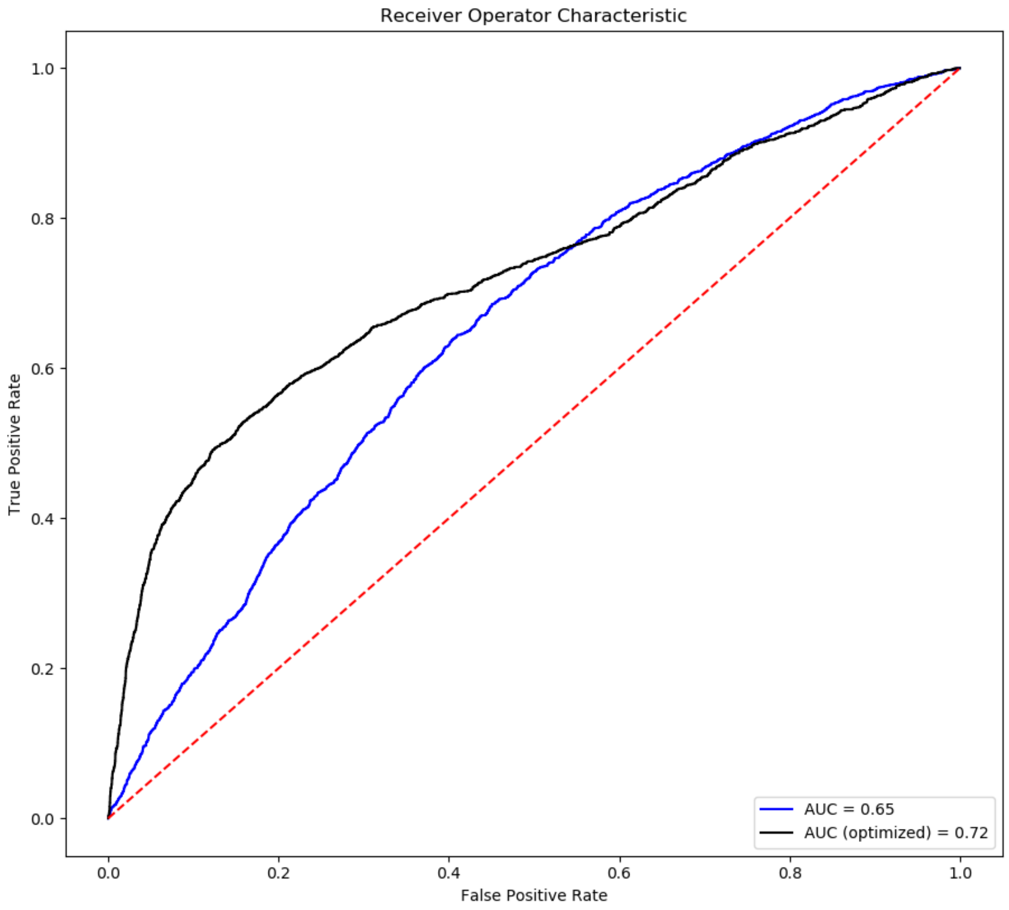
```
# Get ROC curve metrics.
best_fpr, best_tpr, best_threshold = metrics.roc_curve(y_test, best_test_predictions)
best_auc = metrics.auc(best_fpr, best_tpr)
print("Area under the ROC curve: ", best_auc)
```

```
Area under the ROC curve: 0.721368
```

Plot ROC curve for both models

```
# Make an ROC curve plot.
plt.title('Receiver Operator Characteristic')
plt.plot(fpr, tpr, 'blue',
         label = 'AUC = %0.2f'%auc)
plt.plot(best_fpr, best_tpr, 'black',
         label = 'AUC (best) = %0.2f'%best_auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
```

- From the reports, we can see that the AUC and the ROC curve have improved significantly from the base model



Knowledge check 4

Exercise 4

Module completion checklist

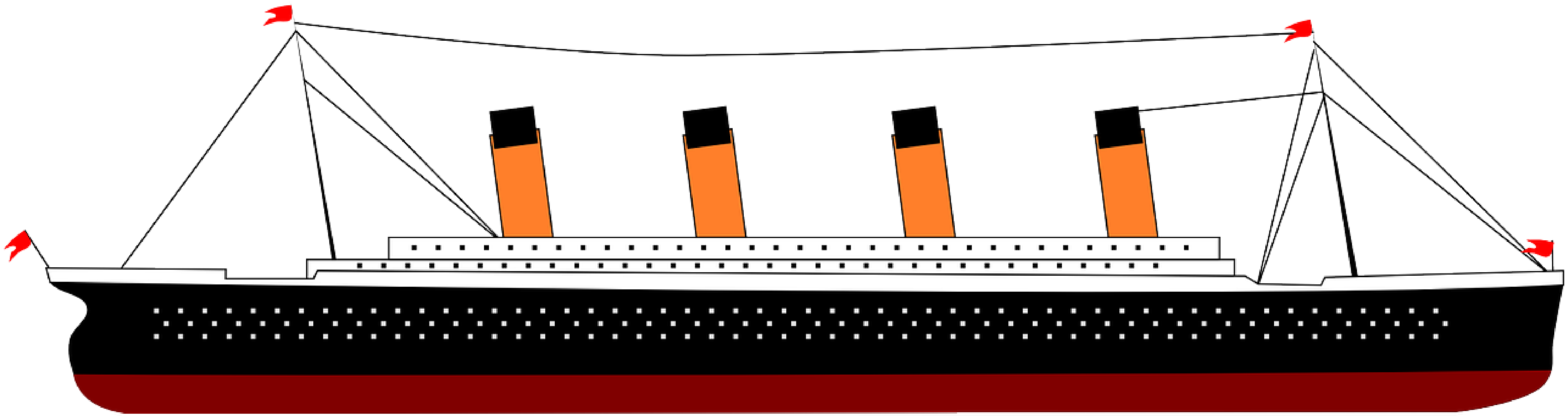
Objective	Complete
Determine when to use logistic regression for classification and transformation of target variable	✔
Summarize the process and the math behind logistic regression	✔
Implement logistic regression on a training dataset and predict on test	✔
Review classification performance metrics and assess results of logistic model performance	✔
Transform categorical variables for implementation of logistic regression	✔
Implement logistic regression on the data and assess results of classification model performance	✔
Analyze the model to determine if / when overfitting occurs	✔
Demonstrate tuning the model using grid search cross-validation	✔

Up next: ensemble methods

- They improve machine learning results by combining several models
- This results in better predictive performance than a single model
- We'll cover two methods: **random forests** and **gradient boosting**

Homework

- Read ***here*** about how one data scientist applied decision trees to the Titanic dataset



This completes our module

Congratulations!