DATA SOCIETY®

Introduction to classification - day 2

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

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	

Directory settings

- In order to maximize the efficiency of your workflow, you should encode your directory structure into `variables`
- Let the `main_dir` be the variable corresponding to your `af-werx` folder

```
# Set `main_dir` to the location of your `af-werx` folder (for Linux).
main_dir = "/home/[username]/Desktop/af-werx"
```

```
# Set `main_dir` to the location of your `af-werx` folder (for Mac).
main_dir = '/Users/[username]/Desktop/af-werx'
```

```
# Set `main_dir` to the location of your `af-werx` folder (for Windows).
main_dir = "C:\\Users\\[username]\\Desktop\\af-werx"
```

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

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
```

Working directory

- Set working directory to `data_dir`

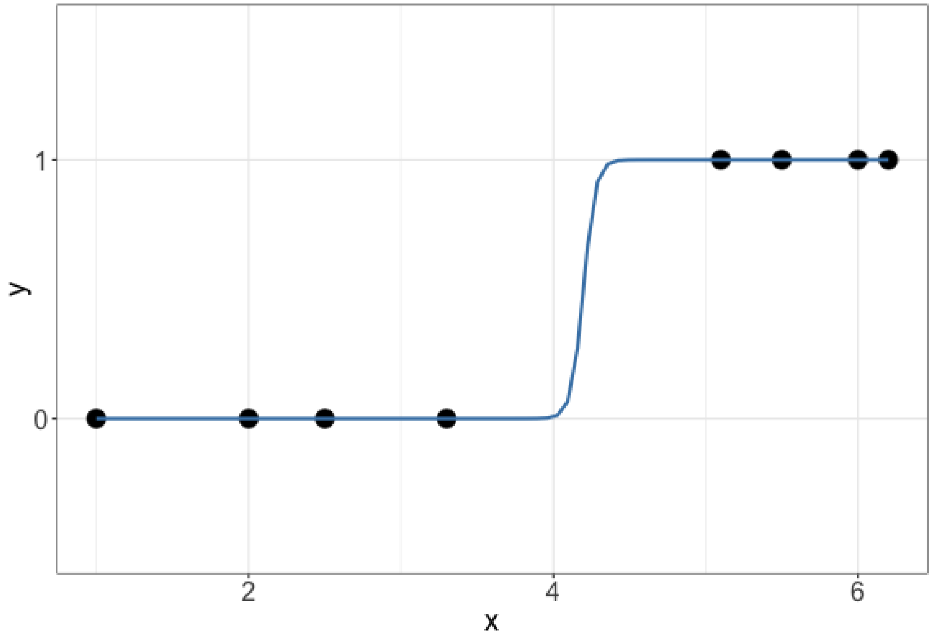
```
# Set working directory.
os.chdir(data_dir)
```

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

```
/home/[user-name]/Desktop/af-werx/data
```

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?

- Logistic regression is a **supervised learning algorithm**
  - We use it to classify data into **categories**
- It outputs **probabilities** and 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 multitudes of **implementations across many programming languages**
  - We can create **robust**, **efficient,** and **well-optimized models**

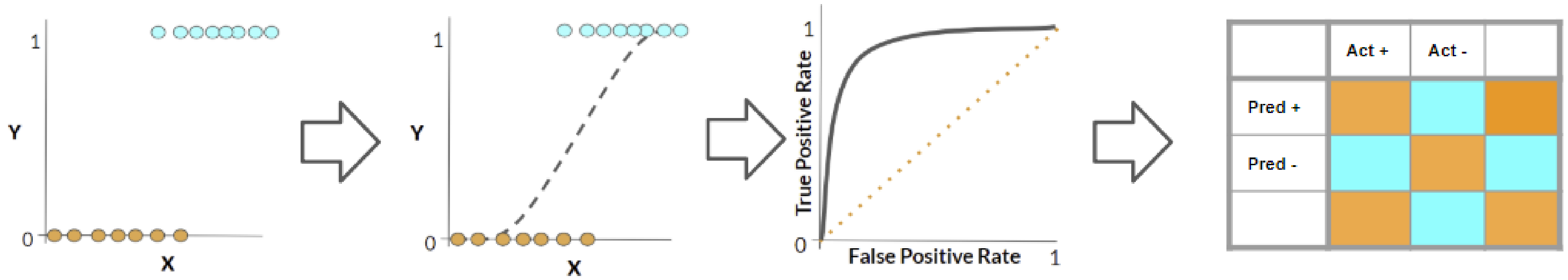
Logistic regression: process

Step 1:
Convert target variable to 1/0

Step 2:
Logistic regression on training data

Step 3:
Use ROC curve & AUC to pick threshold

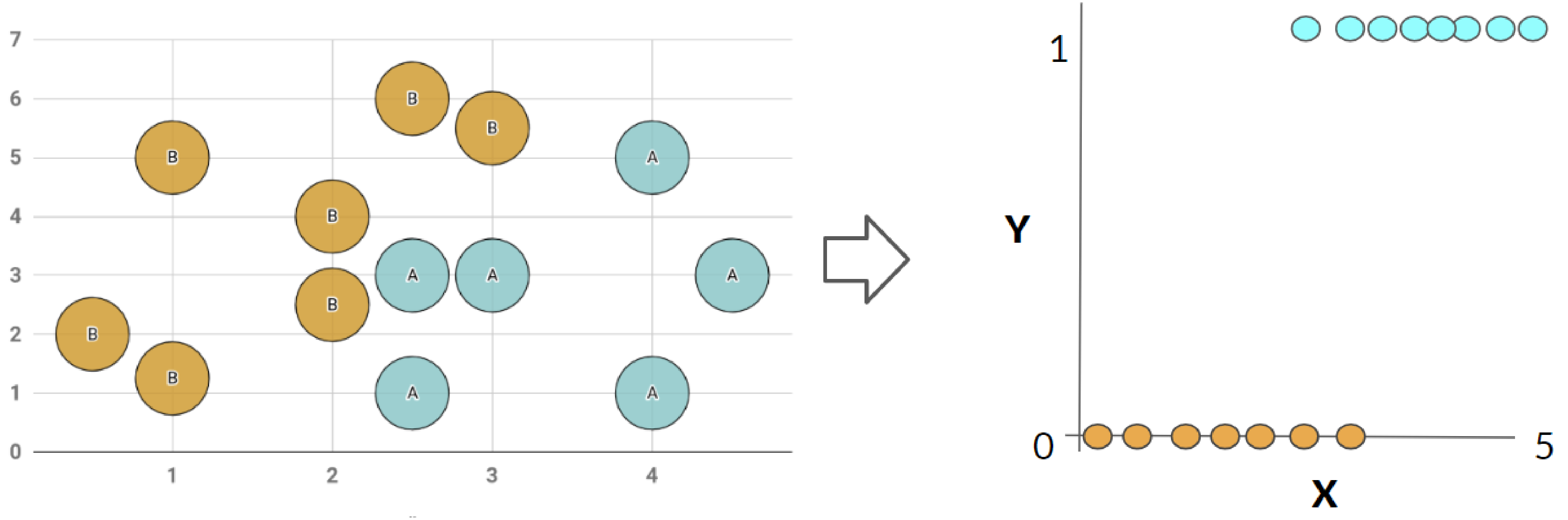
Step 4:
Check performance on test data



Categorical to binary target variable

Two main ways to prepare the target variable:

- **First method:** translate an existing binary variable (i.e. any categorical variable with 2 classes) into 1 and 0



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
  - If the median for the example 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

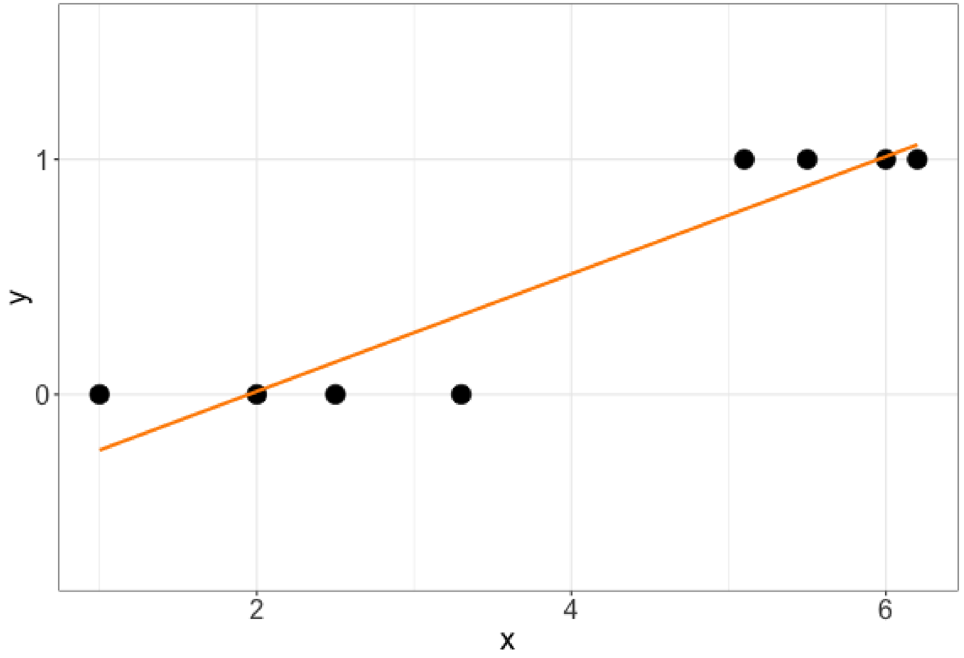
Module completion checklist

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

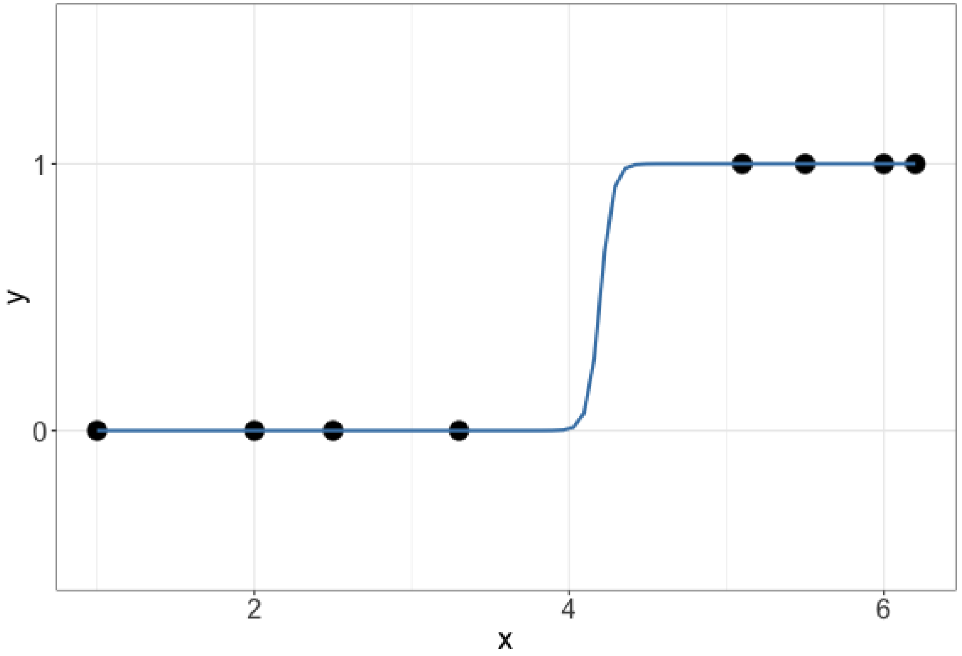
Linear regression line

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

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

- Although it may look a little scary (nobody likes exponents!), we can see a very **familiar equation inside of the parentheses:** $ax + b$

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	

Datasets for logistic regression

- **We will be using two datasets total, we discussed each of the datasets and use cases already**

- **One dataset to learn the concepts in class**
  - **Costa Rica household poverty data**

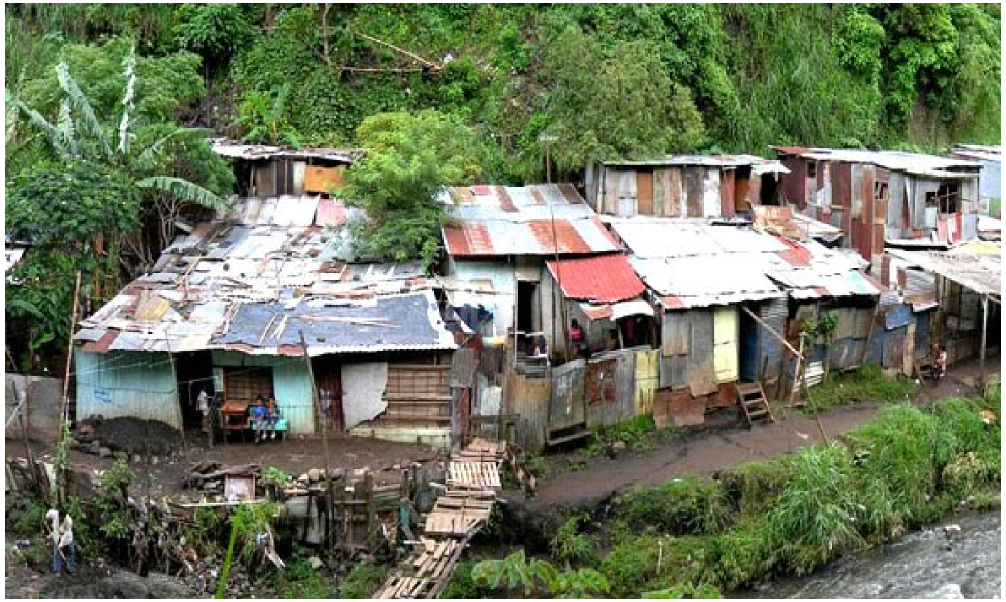
- **One dataset for our in-class exercises**
  - **Chicago census data**

Costa Rican poverty recap

Costa Rican poverty level prediction: proposed solution

- To improve on PMT, the IDB built a competition for Kaggle participants to use methods beyond traditional econometrics

- The given dataset contains Costa Rican household characteristics with a target of four categories:
  - extreme poverty
  - moderate poverty
  - vulnerable households
  - non vulnerable households



Load the dataset

- Let's load the entire dataset

```
household_poverty = pd.read_csv("costa_rica_poverty.csv")
print(household_poverty.head())
```

```
  household_id        ind_id  rooms  ...  age  Target  monthly_rent
0     21eb7fcc1  ID_279628684      3  ...   43       4      190000.0
1     0e5d7a658  ID_f29eb3ddd      4  ...   67       4      135000.0
2     2c7317ea8  ID_68de51c94      8  ...   92       4           NaN
3     2b58d945f  ID_d671db89c      5  ...   17       4      180000.0
4     2b58d945f  ID_d56d6f5f5      5  ...   37       4      180000.0

[5 rows x 84 columns]
```

- The entire dataset consists of `9557` observations and `84` variables

Subsetting data

- In this module, we will run the model on a simple subset
- We don't want to use `monthly_rent` as a variable right now because we had so many NAs
- For our report, your boss wants to see if maybe the **number of rooms** and **number of adults** would predict poverty level well
- Then we are going to predict the same with whole dataset

Subsetting data

- Let's subset our data so that we have 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, along with `monthly_rent`
- Let's name this subset `household_logistic`

```
household_logistic = household_poverty.drop(['household_id', 'ind_id', 'monthly_rent'], axis = 1)
```

- For now, we are only going to use `rooms` and `num_adults` for a simple logistic regression model

The data at first glance

- Look at the data types and the frequency table of the target variable

```
# The data types.
print(household_logistic.dtypes.head())
```

```
rooms             int64
tablet            int64
males_under_12    int64
males_over_12     int64
males_tot         int64
dtype: object
```

```
print(household_logistic['Target'].value_counts())
```

```
4    5996
2    1597
3    1209
1     755
Name: Target, dtype: int64
```

- The target variable is not well-balanced and has **four levels**

Converting the target variable

- Let's convert poverty to a binary target variable, which will help to balance it out
- The levels translate to 1, 2 and 3 as being **vulnerable** households
- Level 4 is **non vulnerable**
- For this reason, we will convert all 1, 2 and 3 to `vulnerable` and 4 to `non_vulnerable`

```
household_logistic['Target'] = np.where(household_logistic['Target'] <= 3, 'vulnerable',
'non_vulnerable')
```

```
print(household_logistic['Target'].head())
```

```
0     non_vulnerable
1     non_vulnerable
2     non_vulnerable
3     non_vulnerable
4     non_vulnerable
Name: Target, dtype: object
```

Data prep: check for NAs

- Check for NAs

```
# Check for NAs.
print(household_logistic.isnull().sum().head())
```

```
rooms              0
tablet             0
males_under_12     0
males_over_12      0
males_tot          0
dtype: int64
```

- We do not have any NAs!

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(household_logistic.dtypes.head())
```

```
rooms             int64
tablet            int64
males_under_12    int64
males_over_12     int64
males_tot         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 `Target`

```
print(household_logistic.Target.dtypes)
```

```
object
```

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

```
household_logistic["Target"] = np.where(household_logistic["Target"] == "non_vulnerable", True, False)

# Check class again.
print(household_logistic.Target.dtypes)
```

```
bool
```

Split into train and test set

- As we did previously, we split our data into training and test sets
- We run logistic regression initially on the training data

```
# Separate predictors from data.
X = household_logistic[['rooms', 'num_adults']]
```

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

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

# Split data into training and test sets, use a 70 test - 30 train 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: 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(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='warn', tol=0.0001, verbose=0,
                   warm_start=False)
```

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

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(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='warn', tol=0.0001, verbose=0,
                   warm_start=False)
```

Logistic regression: predict

The main argument is the same as with most classifiers in `scikit-learn`:

1. `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**

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

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

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	

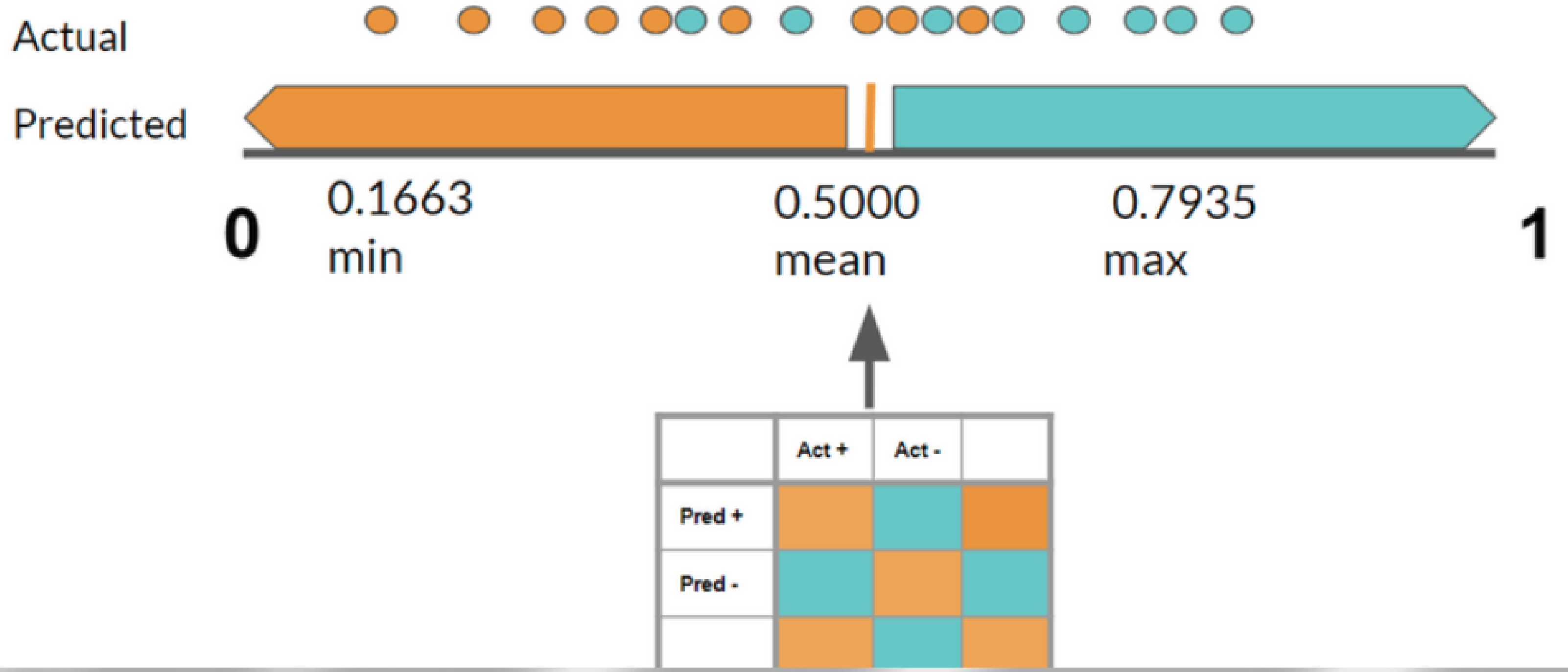
Recap: Confusion matrix

	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 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 `0`
- A typical threshold for logistic regression is `0.5`

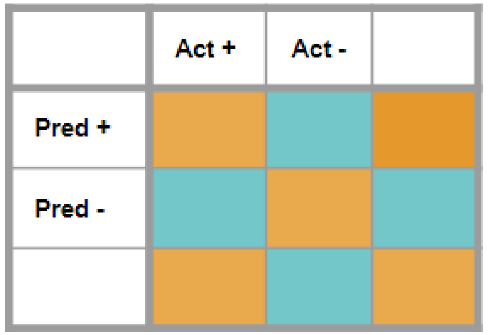


From metrics to a point

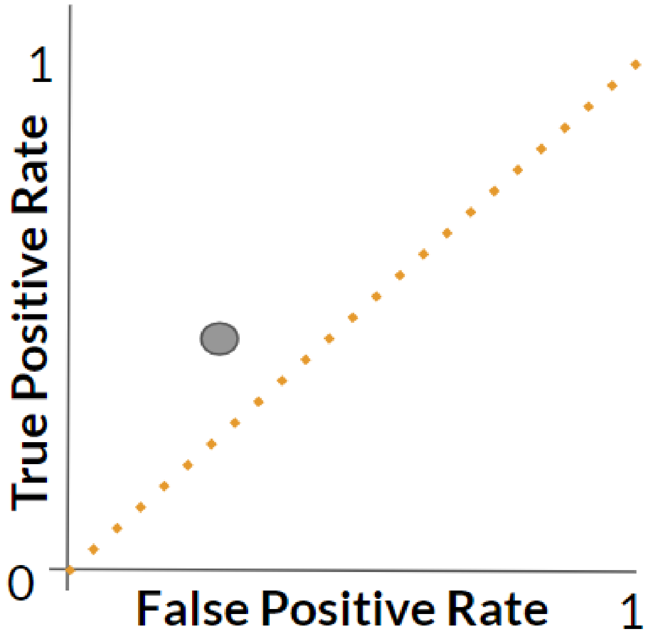
Each threshold can create a confusion matrix, which can be used to calculate a point in space defined by:

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

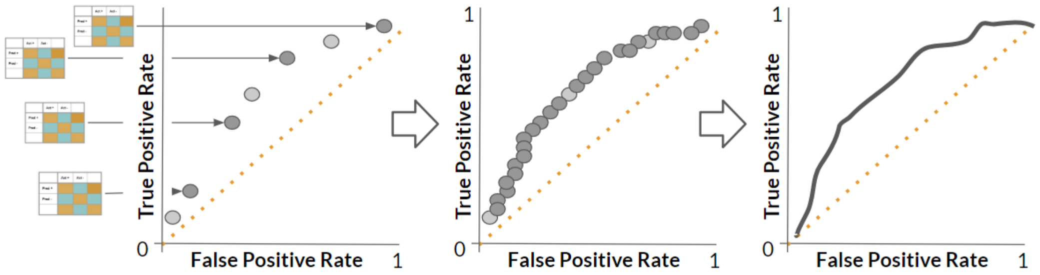
Threshold = 0.50



TPR = 0.42
FPR = 0.32



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

AUC curve:

- It is a performance metric used to compare classification models to measure predictive accuracy
- The AUC should be above .5 to say the model is better than a random guess
- The function to obtain AUC by providing the FPR and TPR is metrics.auc(fpr, tpr)

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 2 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)
```

```
[[ 178  884]
 [ 161 1645]]
```

```
# 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:  0.6356345885634589
```

Classification report

- To make interpretation of the `classification_report` easier, in addition to the 2 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 = ['vulnerable', 'non_vulnerable']
```

```
# 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

    vulnerable       0.53      0.17      0.25      1062
non_vulnerable       0.65      0.91      0.76      1806

      accuracy                           0.64      2868
     macro avg       0.59      0.54      0.51      2868
  weighted avg       0.60      0.64      0.57      2868
```

Classification report (cont'd)

```
print(class_report)
```

```
                precision    recall  f1-score   support

    vulnerable       0.53      0.17      0.25      1062
non_vulnerable       0.65      0.91      0.76      1806

      accuracy                           0.64      2868
     macro avg       0.59      0.54      0.51      2868
  weighted avg       0.60      0.64      0.57      2868
```

- `precision` is **Positive Predictive Value** = **TP** / (**TP** + **FP**)
- `recall` is **TPR** = **TP** / Total positives
- `f1-score` is a weighted harmonic mean of `precision` and `recall`, where it reaches its best value at `1` and worst score at `0`
- `support` is actual number of occurrences of each class in `y_test`

Add accuracy score to the final scores

- So we have it, let's add this score to the dataframe `model_final` that we created in the previous class
- 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' } ,
                                   ignore_index = True)
print(model_final)
```

```
    metrics  values               model
0  accuracy  0.6046               knn_5
1  accuracy  0.6188  knn_GridSearchCV
2  accuracy  0.6287              knn_29
3  accuracy  0.6356            logistic
```

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, :])
```

```
[[0.28499409 0.71500591]
 [0.37610379 0.62389621]
 [0.1624945  0.8375055 ]
 [0.52817721 0.47182279]
 [0.35197966 0.64802034]]
```

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

```
[0.71500591 0.62389621 0.8375055  0.47182279 0.64802034]
```

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.         0.         0.         0.0047081]
```

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

```
True positive:  [0.         0.00387597 0.00609081 0.01052049 0.01162791]
```

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

```
Threshold:  [1.92921126 0.92921126 0.91446334 0.90567607 0.89668599]
```

Computing AUC

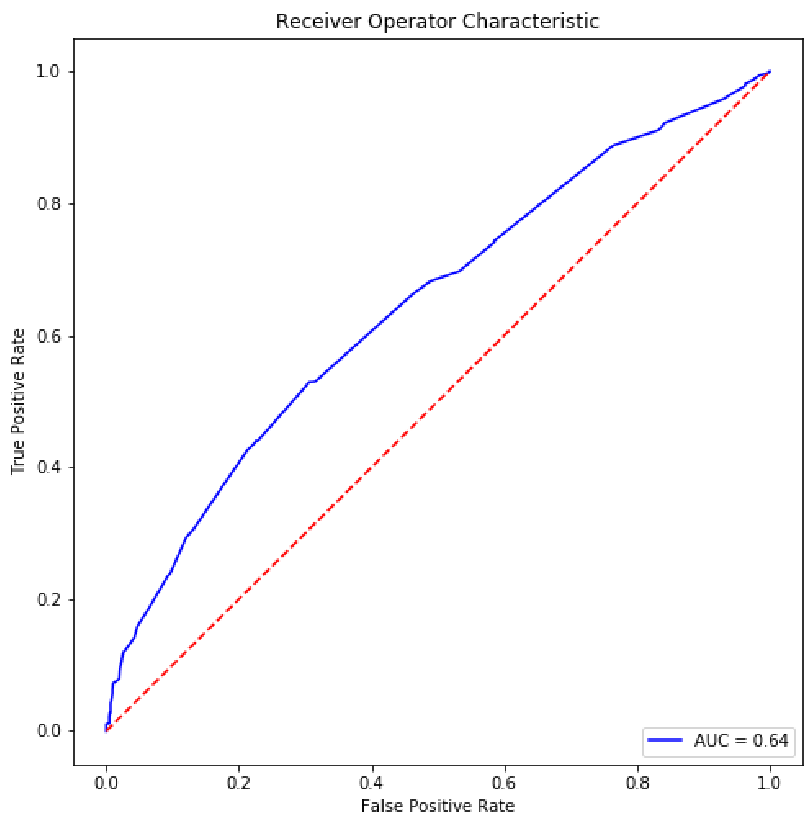
```
# Get AUC by providing the FPR and TPR.
auc = metrics.auc(fpr, tpr)
print("Area under the ROC curve: ", auc)
```

```
Area under the ROC curve:  0.6440758780628705
```

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 `0.635`, which is decent for a base model.
- Our estimated AUC is about `0.644`
- 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(household_logistic.age.head())
```

```
0     43
1     67
2     92
3     17
4     37
Name: age, dtype: int64
```

- Your boss would like for you to convert `age` to a **categorical variable with 3 levels** to analyze varying poverty level between ages

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

Working with categorical variables

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

```
household_logistic.age.value_counts()
```

```
30 or Below            4655
Between 30 and 60      3495
60 and above           1407
Name: age, dtype: int64
```

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

Dummy variables: one hot encoding

- It is 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

Dummy variables: 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` into dummies

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

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

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

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

Transform `age` into dummies

- Let's drop the original `division` column from our Costa Rica subset and concatenate the dummy variables `division_dummy`

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

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

```
   rooms  tablet  males_under_12  ...  Target  60 and above  Between 30 and 60
0      3       0               0  ...    True             0                  1
1      4       1               0  ...    True             1                  0
2      8       0               0  ...    True             1                  0
3      5       1               0  ...    True             0                  0
4      5       1               0  ...    True             0                  1

[5 rows x 82 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 use the whole dataset this time
- We run logistic regression initially on the training data

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

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

```
# 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

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

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

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='warn', tol=0.0001, verbose=0,
                   warm_start=False)
```

- We can see that the default model contains `C = 1` and `penalty = 'l2'`

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(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=None, solver='warn', tol=0.0001, verbose=0,
                   warm_start=False)
```

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)
```

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

Confusion matrix and accuracy

Both `confusion_matrix` and `accuracy_score` take 2 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)
```

```
[[ 687  375]
 [ 243 1563]]
```

```
# 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:  0.7845188284518828
```

Add accuracy score to the final scores

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

```
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  0.6046                   knn_5
1  accuracy  0.6188       knn_GridSearchCV
2  accuracy  0.6287                  knn_29
3  accuracy  0.6356                logistic
4  accuracy  0.7845  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:  0.7806847062341157
```

- 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:
  - Utilize penalization constants and methods to make the model less prone to noise
  - 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 don't
- 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
- These two parameters 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's the role of C?

There are 4 scenarios that might happen with a classifier with respect to 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 C will not "magnify" the effect of the penalty term enough

What's the role of C?

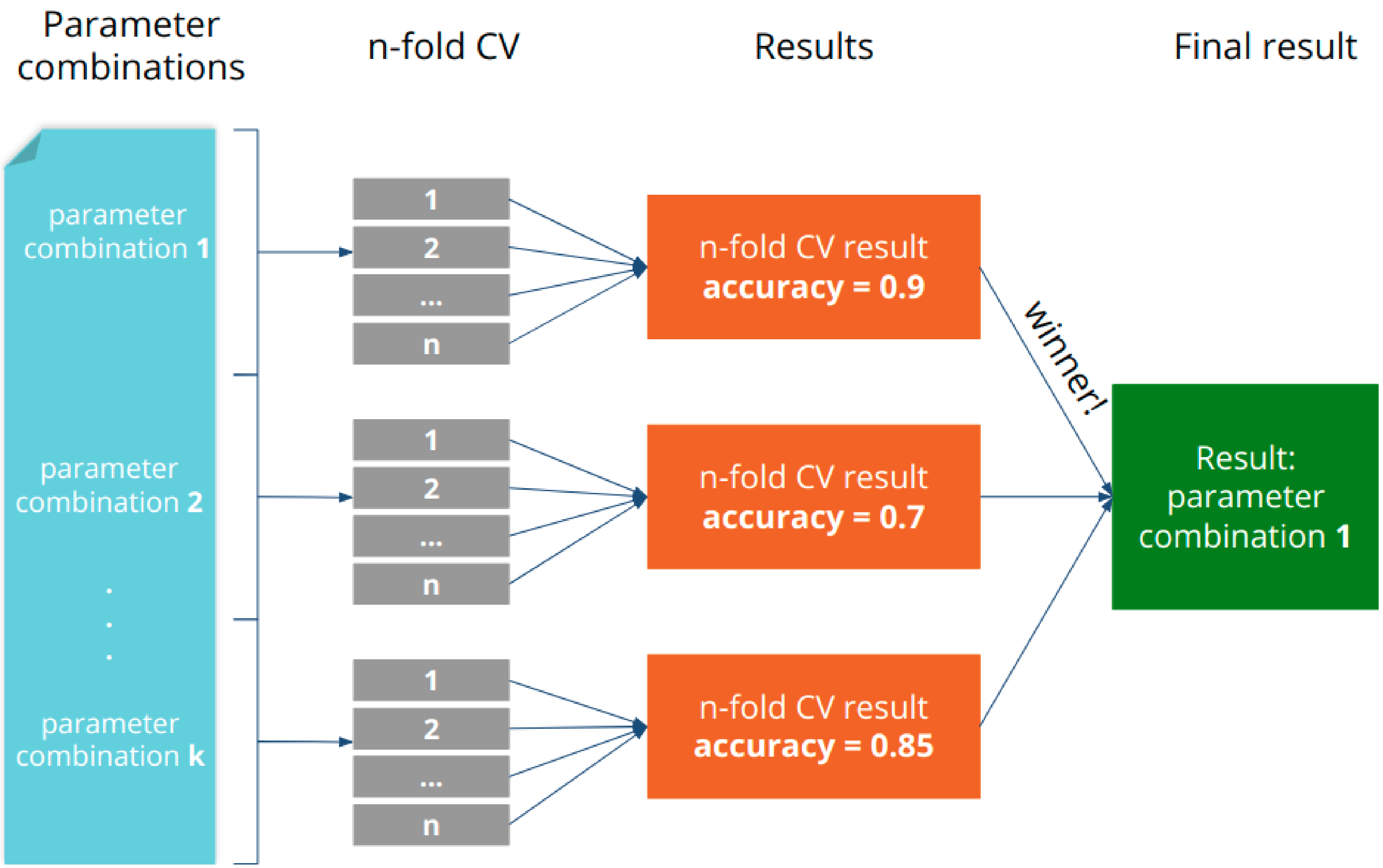
1. $C = large$
   - We run into an **underfitting** problem, where we've weighted and dampened the coefficients too much and we made the model too general
2. $C = optimal$
   - We have a **good, robust, and generalizable model** that works well with new data
   - Ignores most of the noise while preserving 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(), #<- 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, l1_ratio=None,
                                          max_iter=100, multi_class='warn',
                                          n_jobs=None, penalty='l2',
                                          random_state=None, solver='warn',
                                          tol=0.0001, verbose=0,
                                          warm_start=False),
             iid='warn', n_jobs=None,
             param_grid={'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']},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             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:  1.0
```

- It seems like our grid search CV have 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)
```

```
[ True False  True ...  True 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):  0.7859135285913529
```

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)
```

```
[[ 690  372]
 [ 242 1564]]
```

```
# Create a list of target names to interpret class assignments.
target_names = ['Low value', 'High value']
```

```
# 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

   Low value       0.74      0.65      0.69      1062
  High value       0.81      0.87      0.84      1806

    accuracy                           0.79      2868
   macro avg       0.77      0.76      0.76      2868
weighted avg       0.78      0.79      0.78      2868
```

Add accuracy score to the final scores

- Let's add this accuracy score to the dataframe `model_final`

```
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  0.6046                  knn_5
1  accuracy  0.6188       knn_GridSearchCV
2  accuracy  0.6287                 knn_29
3  accuracy  0.6356               logistic
4  accuracy  0.7845  logistic_whole_dataset
5  accuracy  0.7859         logistic_tuned
```

```
pickle.dump(model_final, open("model_final_logistic.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, ])
```

```
[[0.0446763  0.9553237 ]
 [0.82715447 0.17284553]
 [0.28190006 0.71809994]
 [0.35725272 0.64274728]
 [0.01030444 0.98969556]]
```

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

```
[0.9553237  0.17284553 0.71809994 0.64274728 0.98969556]
```

Get metrics for ROC curve (cont'd)

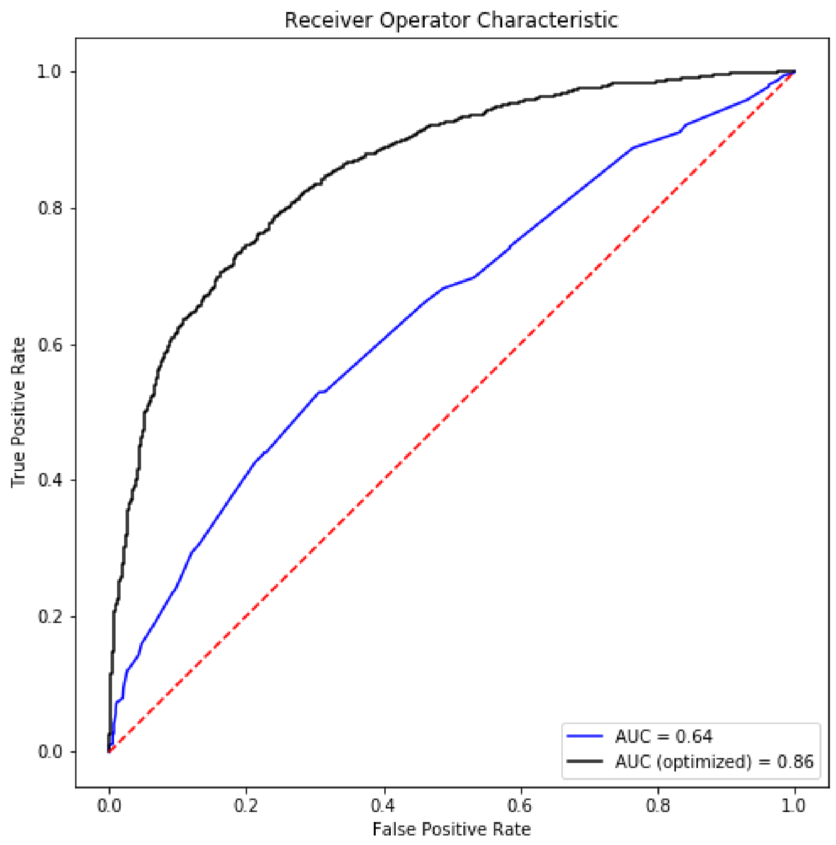
```
# Get ROC curve metrics.
best_fpr, best_tpr, best_threshold = metrics.roc_curve(y_test, best_test_predictions)
best_auc = metrics.auc(best_fpr, best_tpr)
print(best_auc)
```

```
0.8556277151074155
```

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	✔

Workshop!

- Workshops are to be completed in the afternoon either with a dataset for a capstone project or with another dataset of your choosing
- Make sure to annotate and comment your code so that it is easy for others to understand what you are doing
- This is an exploratory exercise to get you comfortable with the content we discussed today
- Today you will:
  - Use the logistic regression model to classify your data
  - Prepare and do the exploratory data analysis before the modeling
  - Check if over fitting occurs and how the model compares to kNN model
  - Implement grid search cross validation to avoid overfitting

This completes our module

Congratulations!