DATA SOCIETY®

Introduction to Classification - Day 1

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

Module completion checklist

Objective	Complete
Understanding classification and it's uses	
Summarize steps & application of kNN	
Clean and transform data to run kNN	
Define cross-validation and how and when it is used	
Implement kNN algorithm on the training data without cross-validation	
Identify performance metrics for classification algorithms and evaluate simple kNN model	
Find optimal hyperparameters with a grid search	
Use GridSearchCV to find optimal number of nearest neighbors and define optimal model	
Discuss reasons we would or would not use kNN	

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

Let's load the packages we will be using:

```
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt


# New today - we will introduce it when we use it
import pickle
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import scale
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn import metrics
```

Working directory

- Set working directory to the `data_dir` variable we set
- We do this using the `os.chdir` function, change directory
- We can then check the working directory using `.getcwd()`
- For complete documentation of the `os` package, ***click here***

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

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

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

Classification vs. regression

	Classification	Regression
Target variable	Discrete, usually binary	Continuous
Types	Binary, Multi-Class	Linear
Algorithms	Decision trees, random forest, logistic regression, K-nearest neighbors	Linear regression, regression trees, time-series regression
Type of learning method	Supervised - categorical target	Supervised - continuous target

Classification: use cases

- **Below are some examples of how you would apply classification algorithms in business scenarios**

Question to answer	Real world example
What is this object like?	Selecting similar products at the lowest prices
Who is this person like?	Anticipating behavior or preferences of a person based on her similarities with others
What category is this in?	Anticipate if your customer is pregnant, remodeling, just got married, etc.
What is the probability that something is in a given category?	Determine the probability that a piece of equipment will fail, determine the probability that someone will buy your product

Module completion checklist

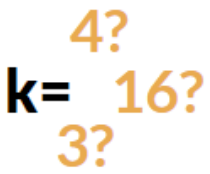
Objective	Complete
Understanding classification and it's uses	✔
Summarize steps & application of kNN	
Clean and transform data to run kNN	
Define cross-validation and how and when it is used	
Implement kNN algorithm on the training data without cross-validation	
Identify performance metrics for classification algorithms and evaluate simple kNN model	
Find optimal hyperparameters with a grid search	
Use GridSearchCV to find optimal number of nearest neighbors and define optimal model	
Discuss reasons we would or would not use kNN	

kNN : what is it?

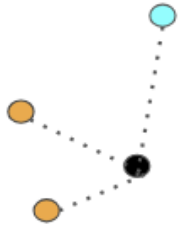
- The k-Nearest Neighbors (kNN) algorithm is a **supervised** algorithm
- It is primarily used for **classification**
- It takes a bunch of **labeled points** and uses them to learn how to label other points
- It is based on an algorithm that involves distance calculation

Steps of kNN

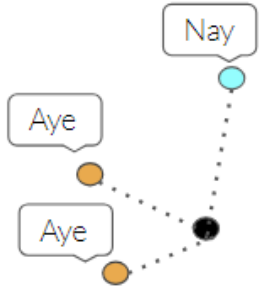
Step 1: Select K



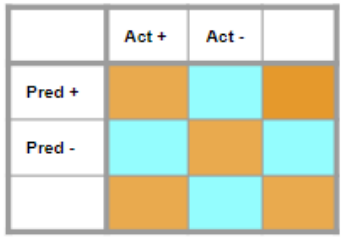
Step 2: Measure Distance



Step 3: Majority Vote

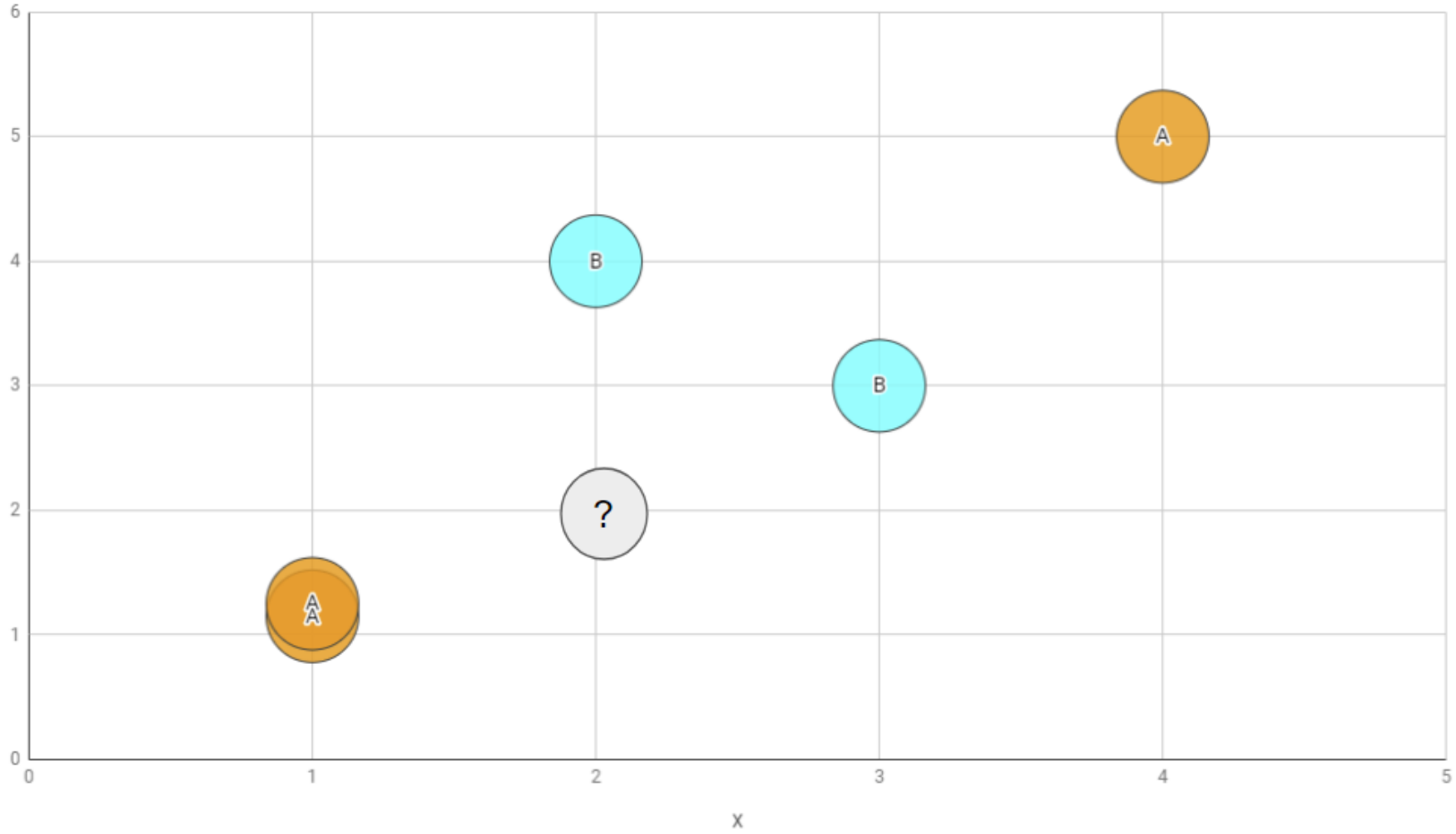


Step 4: Check Performance Metrics

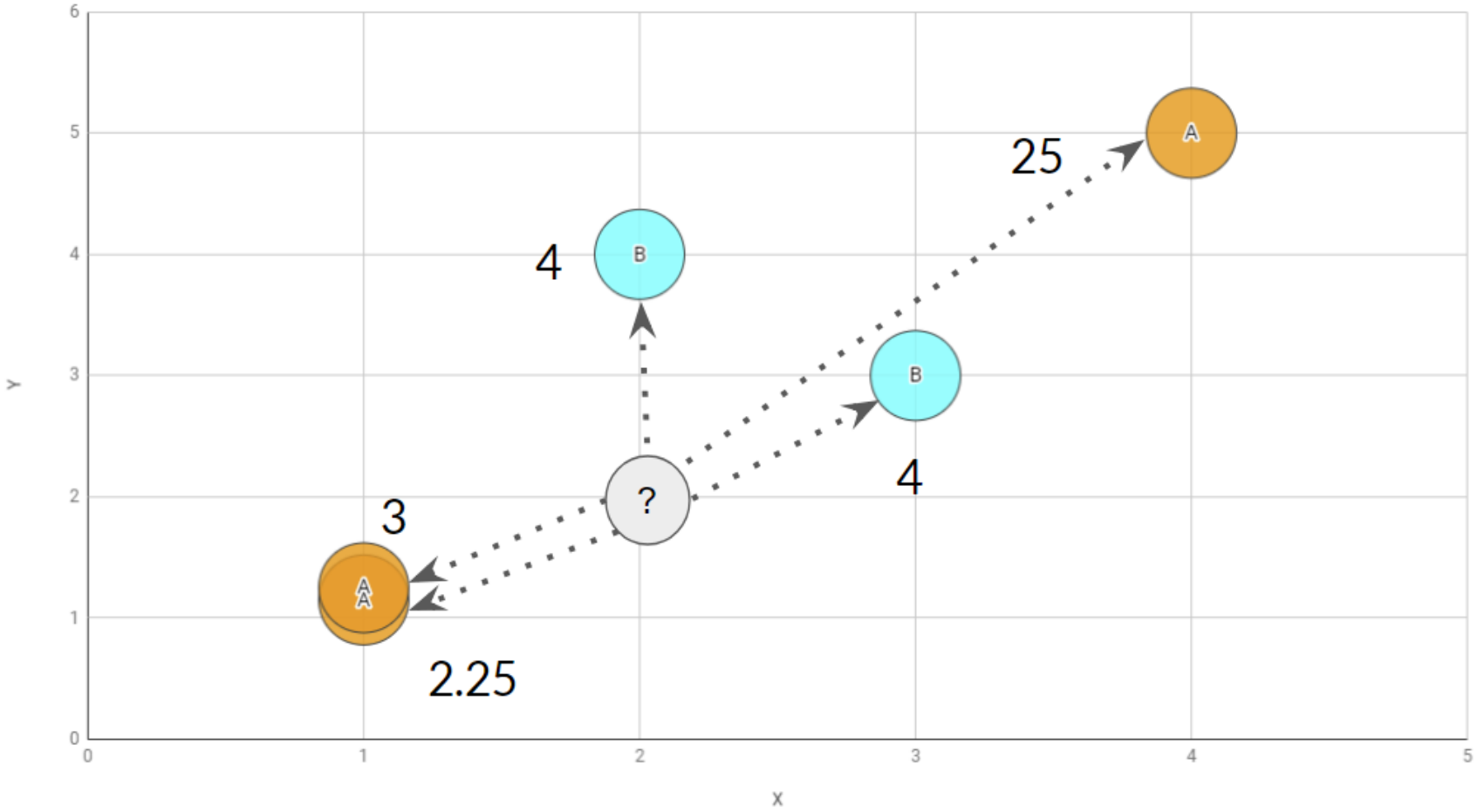


Step 5: Find Best K

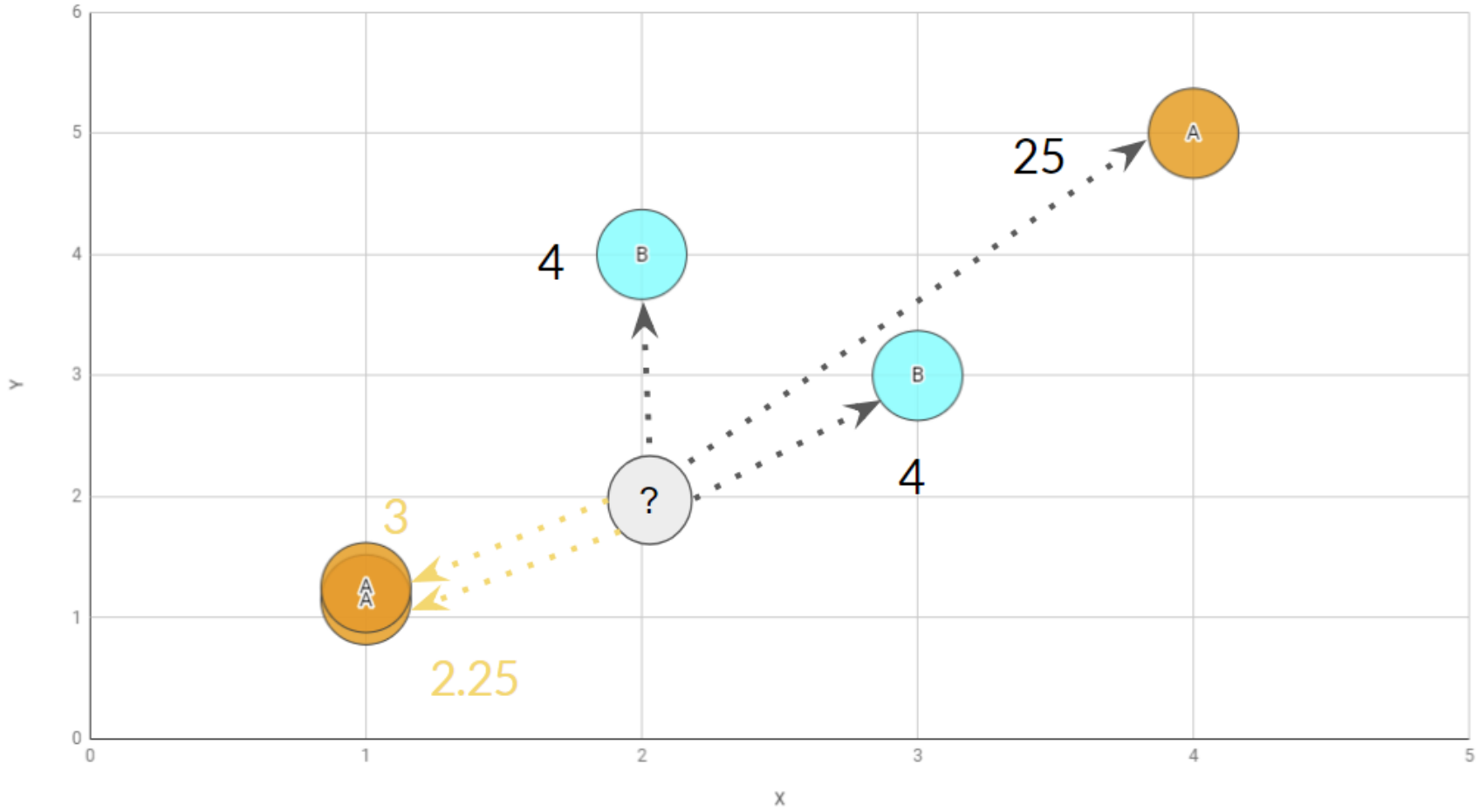
k-Nearest Neighbors: setup



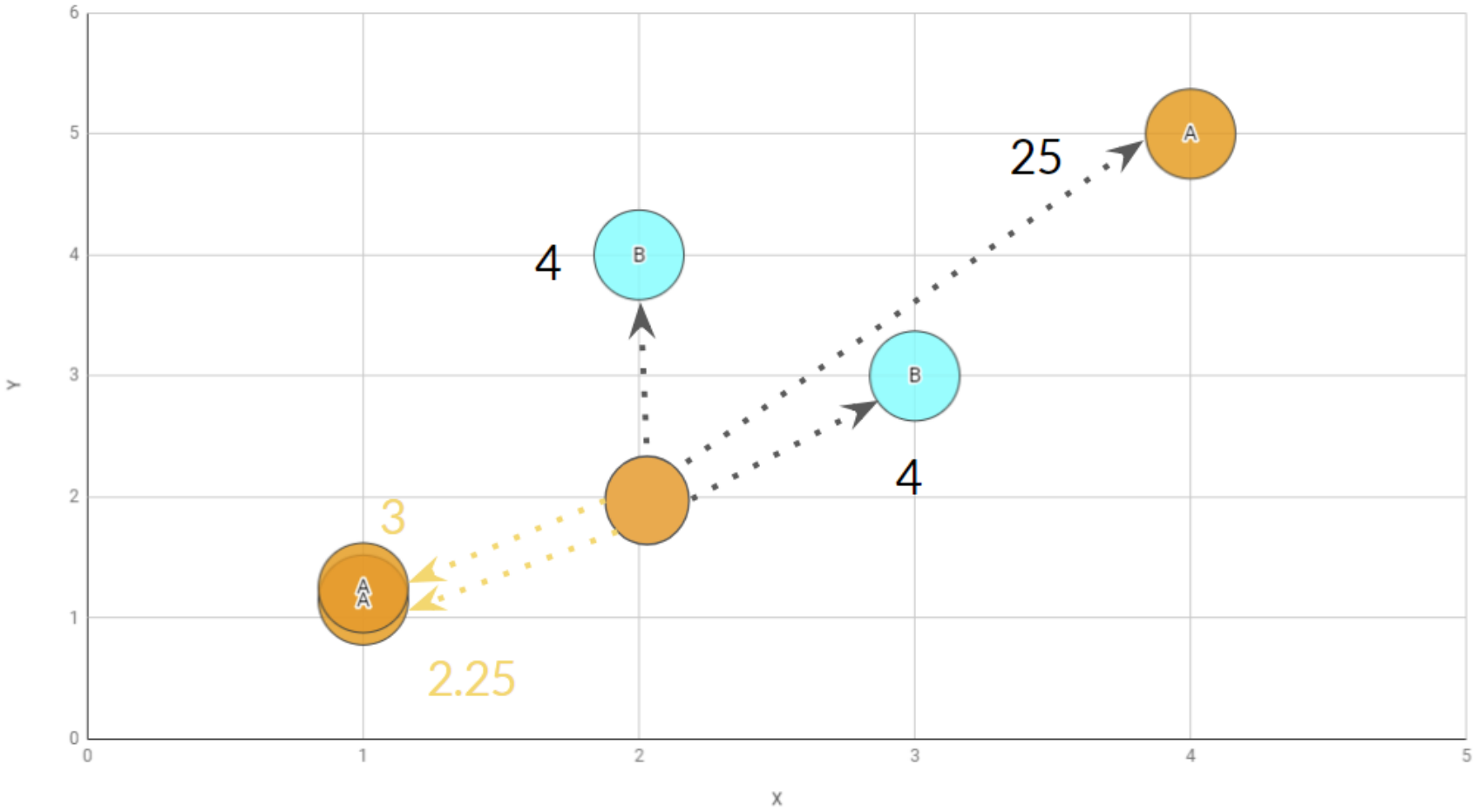
k-Nearest Neighbors: measure



k-Nearest Neighbors: 2-NN for majority vote



k-Nearest Neighbors: label point



Module completion checklist

Objective	Complete
Understanding classification and it's uses	✔
Summarize steps & application of kNN	✔
Clean and transform data to run kNN	
Define cross-validation and how and when it is used	
Implement kNN algorithm on the training data without cross-validation	
Identify performance metrics for classification algorithms and evaluate simple kNN model	
Find optimal hyperparameters with a grid search	
Use GridSearchCV to find optimal number of nearest neighbors and define optimal model	
Discuss reasons we would or would not use kNN	

Datasets for kNN

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

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

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

Costa Rican poverty: case study

- We already have been introduced to the case study from the **Inter-American Development Bank (IDB)**
- For a quick refresher: recently, the **IDB** conducted a competition amongst data scientists on ***Kaggle.com***

Costa Rican poverty: backstory

Costa Rican poverty level prediction

As stated by the **IDB**:

- Social programs have a hard time making sure the right people are given enough aid
- It's especially tricky when a program focuses on the poorest segment of the population
- The world's poorest typically can't provide the necessary income and expense records to prove that they qualify

Costa Rican poverty: backstory

Proxy Means Test (PMT)

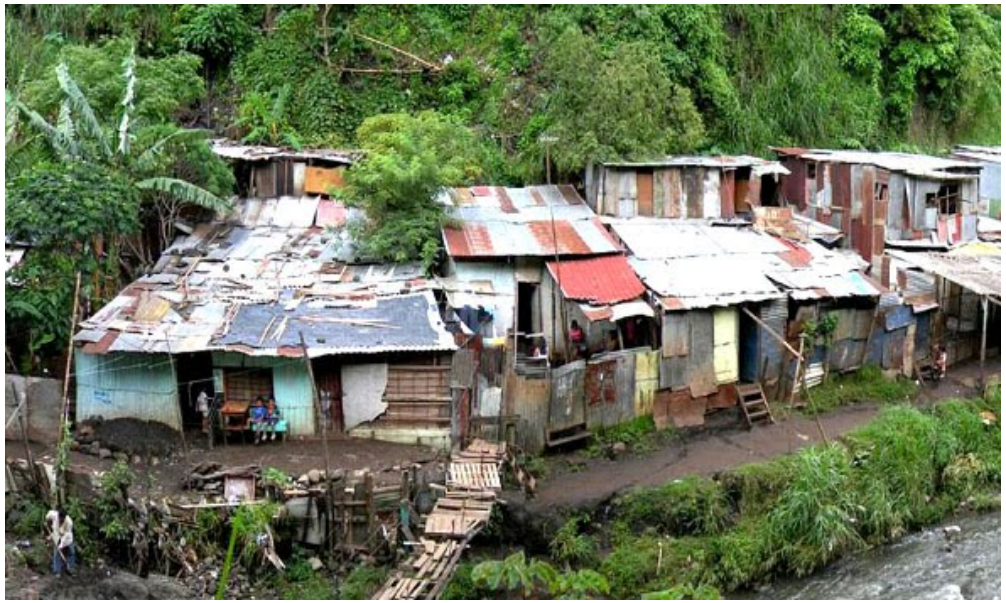
- In Latin America, one popular method uses an algorithm to verify income qualification
- It's called the **Proxy Means Test (or PMT)**
- With the PMT, agencies use a model that considers a family's observable household attributes like the material of their walls and ceiling, or the assets found in the home, to classify them and predict their level of need
- While this is an improvement, accuracy remains a problem as the region's population grows and poverty declines

Costa Rican poverty: proposed solution

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



Our goal with this dataset

- Understand the patterns and groups within the dataset
- Predict poverty levels of Costa Rican households
- Build a model that is reproducible for other countries as well

Predicting poverty - kNN

- Today, we will be using kNN to predict poverty
- Because kNN works much better with fewer dimensions, we will be taking a small subset of the actual dataset
- As we move towards more complex machine learning algorithms, we will add more variables

Loading data into Python

- Let's load the entire dataset
- For clustering, we will be taking a specific subset
- We are now going to use the function `read_csv` to read in our `household_poverty` 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 once again subset data, however this time we will use some new variables:
  - **household id**
  - **rooms**
  - **num_adults**
  - *Target*
- We don't want to use `monthly_rent` as a variable right now because we had so many NAs
- We want to see if maybe the **number of rooms** and **number of adults** would predict poverty level well

Subsetting data

- Let's subset our data so that we have the variables we need for building `kNN`
- Once again, we are keeping `household_id`, `rooms`, `num_adults`, and `Target`
- Let's name this subset `costa_knn`

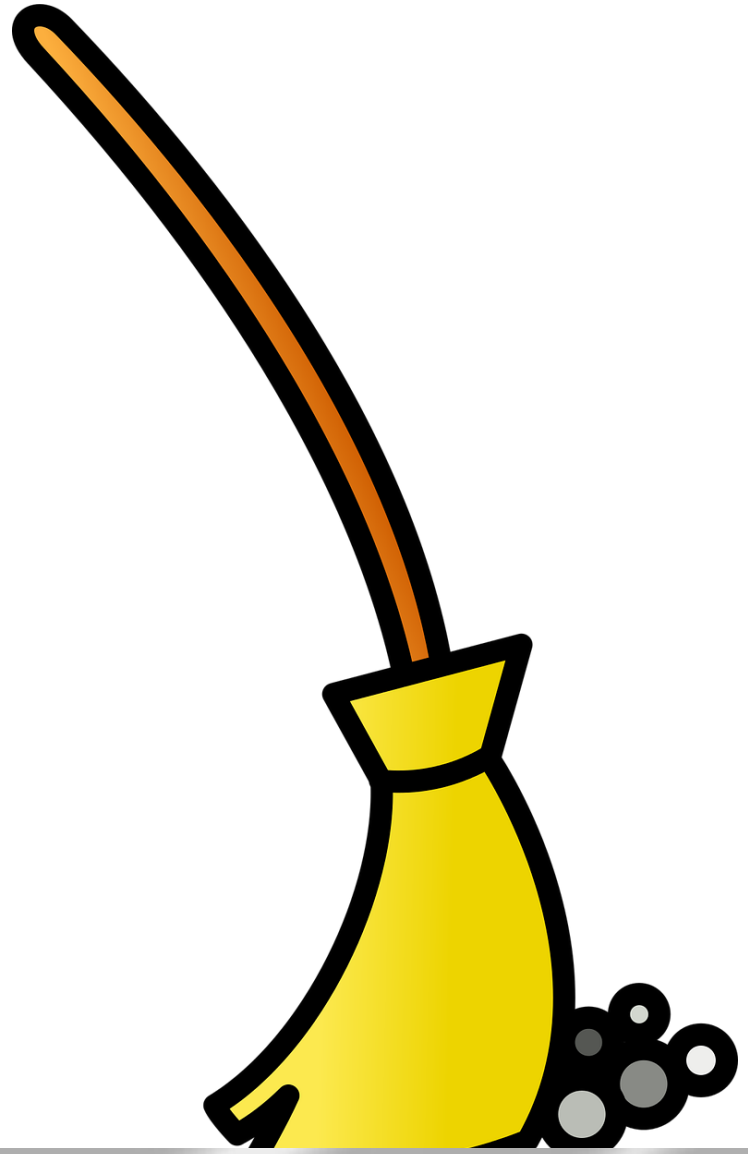
```
costa_knn = household_poverty[["household_id","rooms","num_adults","Target"]]
print(costa_knn.head())
```

```
  household_id  rooms  num_adults  Target
0    21eb7fcc1      3           1       4
1    0e5d7a658      4           1       4
2    2c7317ea8      8           1       4
3    2b58d945f      5           2       4
4    2b58d945f      5           2       4
```

Data cleaning steps for kNN

- There are a few steps to remember to take before splitting the data and training the model
- We have already learned how to check for NAs
- We will learn more about these steps in the upcoming slides

1. Make sure the target is labeled
2. Check for NAs
3. Scale the predictors



The data at first glance

- Look at the first 5 rows and the data types

```
# The first 5 rows.
print(costa_knn.head())
```

```
  household_id  rooms  num_adults  Target
0     21eb7fcc1      3           1       4
1     0e5d7a658      4           1       4
2     2c7317ea8      8           1       4
3     2b58d945f      5           2       4
4     2b58d945f      5           2       4
```

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

```
household_id    object
rooms            int64
num_adults       int64
Target           int64
dtype: object
```

- Frequency table of the target variable

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

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

- The target variable is not well balanced
- It also has **four levels**, we are going to make it binary for now
- This will also help balance it out

Converting the target variable

- Let's convert poverty to a binary target variable
- 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`

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

```
print(costa_knn['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(costa_knn.isnull().sum())
```

```
household_id    0
rooms           0
num_adults      0
Target          0
dtype: int64
```

- We do not have any NAs, we are now ready to scale our predictors!

Data prep: numeric variables

- In `kNN`, we 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(costa_knn.dtypes)
```

```
household_id    object
rooms            int64
num_adults       int64
Target          object
dtype: object
```

Data prep: ready for kNN

- 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(costa_knn.Target.dtypes)
```

```
object
```

- We want to convert this to `bool` so that is is a binary class

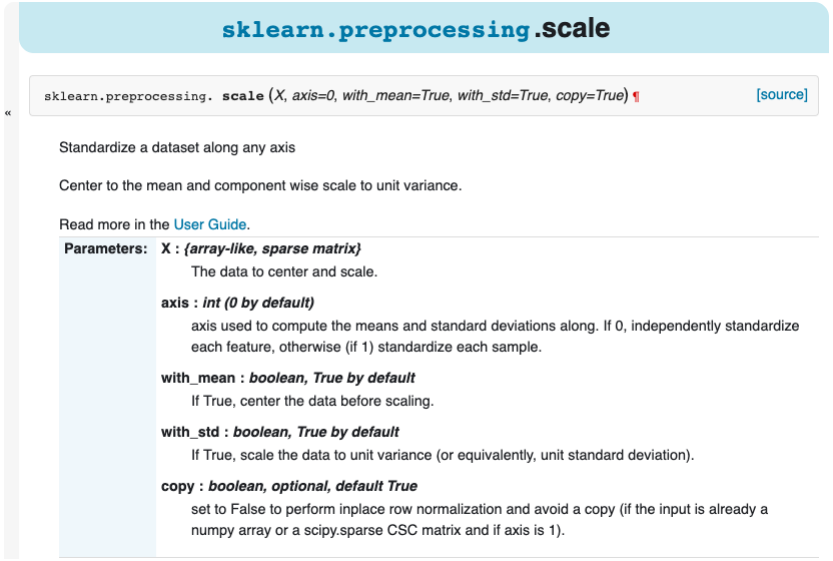
```
costa_knn["Target"] = np.where(costa_knn["Target"] == "non_vulnerable", True, False)

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

```
bool
```

- **Success! Now let's scale our predictors**

Data prep: scaling variables

- Once the data is converted to `numeric` (if necessary), we **scale** the dataset
- There are a few methods to scale data and we will use the `scale` function from `sklearn.preprocessing`:
- A few things to remember about `scale`:
  - it is a generic function whose default method **centers** and/or scales the columns of a numeric matrix
  - it will convert your dataset to have a `mean` of `0` and a `standard deviation` of `1`



sklearn.preprocessing.scale

`sklearn.preprocessing.` **`scale`** (*X, axis=0, with_mean=True, with_std=True, copy=True*) [source]

Standardize a dataset along any axis

Center to the mean and component wise scale to unit variance.

Read more in the User Guide.

Parameters:

X : ***{array-like, sparse matrix}***
The data to center and scale.

axis : ***int (0 by default)***
axis used to compute the means and standard deviations along. If 0, independently standardize each feature, otherwise (if 1) standardize each sample.

with_mean : ***boolean, True by default***
If True, center the data before scaling.

with_std : ***boolean, True by default***
If True, scale the data to unit variance (or equivalently, unit standard deviation).

copy : ***boolean, optional, default True***
set to False to perform inplace row normalization and avoid a copy (if the input is already a numpy array or a scipy.sparse CSC matrix and if axis is 1).

Data prep: scaling variables

- To scale only our predictors, we split our data into `X` and `y`
- We did this last week, when we split our data into training and test sets

```
# Split the data into X and y - y is categorical, so can't scale.
X = costa_knn[['rooms', 'num_adults']]
y = np.array(costa_knn['Target'])

# Scale X.
X_scaled = scale(X)
print(X_scaled[0:5])
```

```
[[-1.33182893 -1.3657179 ]
 [-0.65077114 -1.3657179 ]
 [ 2.07346003 -1.3657179 ]
 [ 0.03028665 -0.5080948 ]
 [ 0.03028665 -0.5080948 ]]
```

Knowledge Check 1

Exercise 1

Module completion checklist

Objective	Complete
Understanding classification and it's uses	✔
Summarize steps & application of kNN	✔
Clean and transform data to run kNN	✔
Define cross-validation and how and when it is used	
Implement kNN algorithm on the training data without cross-validation	
Identify performance metrics for classification algorithms and evaluate simple kNN model	
Find optimal hyperparameters with a grid search	
Use GridSearchCV to find optimal number of nearest neighbors and define optimal model	
Discuss reasons we would or would not use kNN	

Introducing cross-validation

- We need to split the data into a **training set** and a **test set** like we did last week
- But now, we are doing this **multiple times**
- We have a new **test set** for each fold `n`
- The rest of the data is the **training set**

Why do we use cross-validation?

- Cross-validation is helpful in multiple ways:
  - **It tunes our model better by running it multiple times on our data instead of just once on the training set and once on the test set**
  - **You get assurance that your model has got most of the patterns from the data correct and it's not picking up too much on the noise**
  - **It finds optimal parameters for your model because it runs multiple times**

Cross-validation: train and test

Train

- This is the data that you **train your model on**
- Usually about **70% of your dataset**
- Use a larger portion of the data to train so that the model gets a **large enough sample of the population**
- **When there is not a large population on the whole, cross-validation techniques can be implemented**

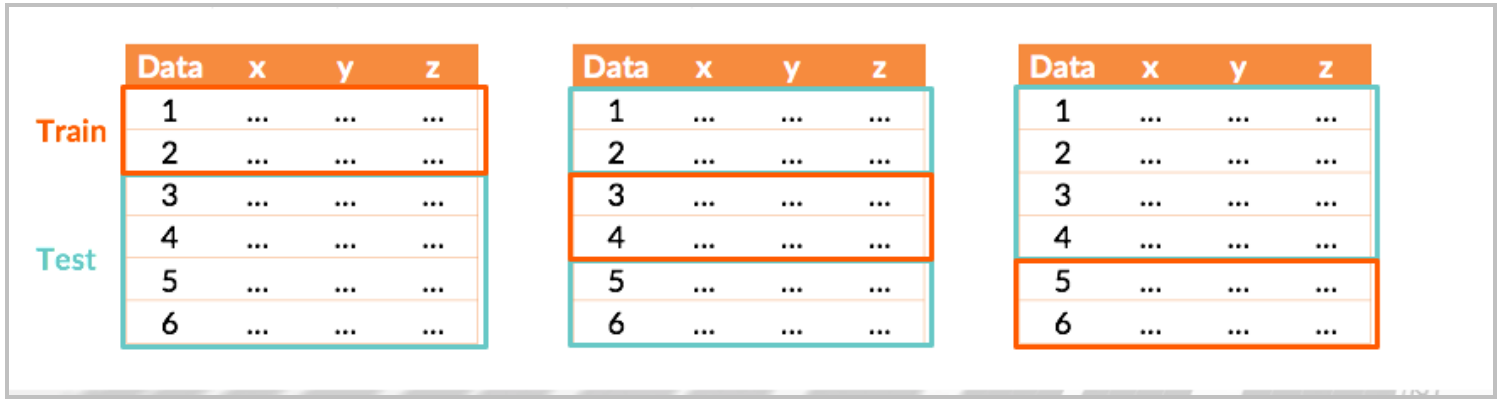
Test

- This is the data that you **test your model on**
- Usually about **30% of your dataset**
- Use a smaller portion to test your trained model on
- **When cross-validation is implemented, small test sets will be held out multiple times**

Cross-validation: n-fold

Here is how cross-validation works:

1. Split the dataset into several subsets ("n" number of subsets) of equal size
2. **Use each subset as the test dataset** and **use the rest of the data as the training dataset**
3. Repeat the process for every subset you create



Train & test: small scale before n-fold

- Before we actually use n-fold cross-validation:
  - We split our data into a train and test set
  - We run kNN initially on the training data

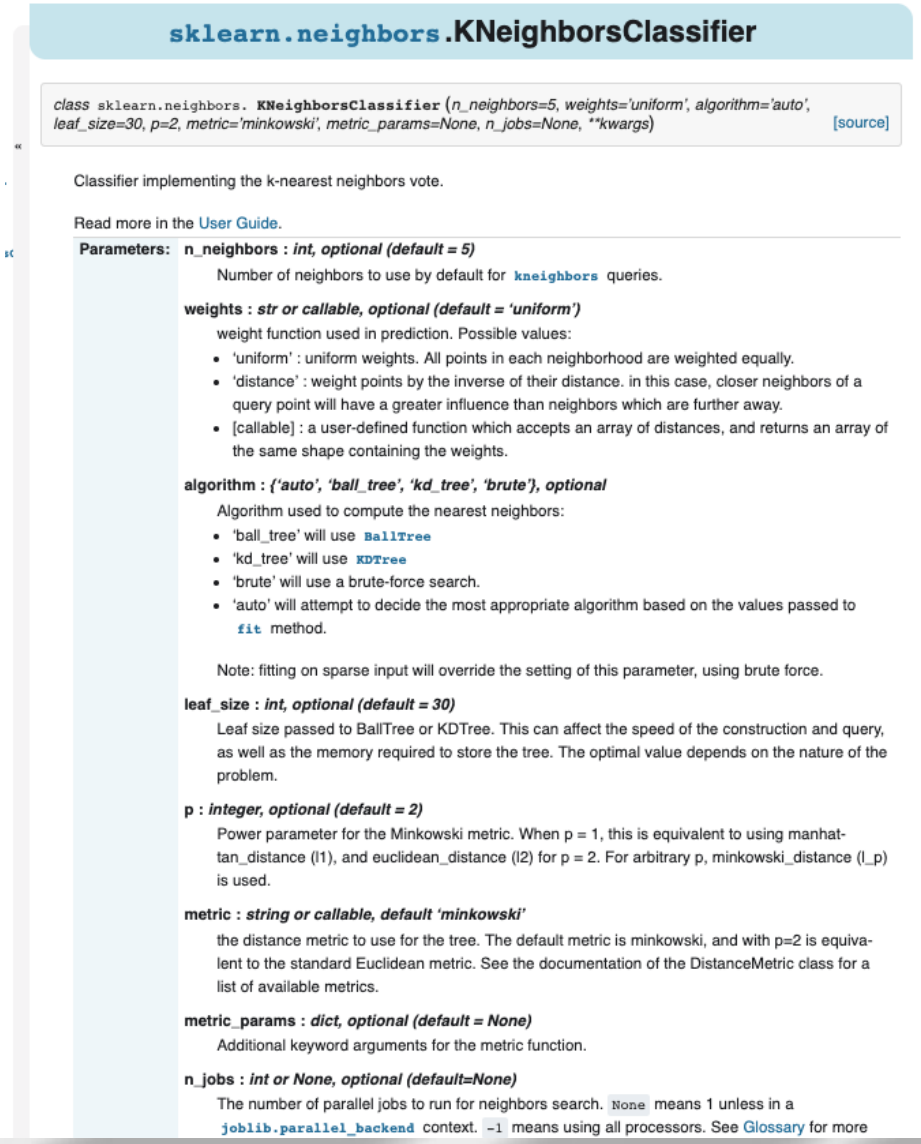
```
# Set the seed.
np.random.seed(1)

# Split into train and test.
X_train, X_test, y_train, y_test = train_test_split(X_scaled,
                                                    y,
                                                    test_size = 0.3)
```

Module completion checklist

Objective	Complete
Understanding classification and it's uses	✔
Summarize steps & application of kNN	✔
Clean and transform data to run kNN	✔
Define cross-validation and how and when it is used	✔
Implement kNN algorithm on the training data without cross-validation	
Identify performance metrics for classification algorithms and evaluate simple kNN model	
Find optimal hyperparameters with a grid search	
Use GridSearchCV to find optimal number of nearest neighbors and define optimal model	
Discuss reasons we would or would not use kNN	

kNN: modeling with KNeighborsClassifier

- We will use `sklearn.neighbors` function, `KNeighborsClassifier`
- We will be using mostly `sklearn` modules and functions for classification and machine learning



sklearn.neighbors.KNeighborsClassifier

class sklearn.neighbors. **KNeighborsClassifier** (*n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None, **kwargs*) [source]

Classifier implementing the k-nearest neighbors vote.

Read more in the User Guide.

Parameters:

n_neighbors : *int, optional (default = 5)*
Number of neighbors to use by default for `kneighbors` queries.

weights : *str or callable, optional (default = 'uniform')*
weight function used in prediction. Possible values:
- 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
- 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

algorithm : *{'auto', 'ball_tree', 'kd_tree', 'brute'}, optional*
Algorithm used to compute the nearest neighbors:
- 'ball_tree' will use `BallTree`
- 'kd_tree' will use `KDTree`
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit` method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

leaf_size : *int, optional (default = 30)*
Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

p : *integer, optional (default = 2)*
Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric : *string or callable, default 'minkowski'*
the distance metric to use for the tree. The default metric is minkowski, and with p=2 is equivalent to the standard Euclidean metric. See the documentation of the DistanceMetric class for a list of available metrics.

metric_params : *dict, optional (default = None)*
Additional keyword arguments for the metric function.

n_jobs : *int or None, optional (default=None)*
The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See Glossary for more

kNN: build model

- We now will instantiate our kNN model and run it on `X_train`
- At first, we will simply run the model on our training data and predict on test
- We set `n_neighbors = 5` as a random guess, usually we can use 3 or 5
- Next week, we will use cross-validation to optimize our model
- Using this process, we will also choose the best `n_neighbors` for an optimal result

```
# Create KNN classifier.
knn = KNeighborsClassifier(n_neighbors = 5)
# Fit the classifier to the data.
knn.fit(X_train, y_train)
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                     weights='uniform')
```

kNN: predict on test

- Now we will take our trained model and predict on the test set

```
predictions = knn.predict(X_test)
```

- What we get is a vector of predicted values

```
print(predictions[0:5])
```

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

kNN: predict on test

- **This is helpful because we have the actual values for this sample**
- Let's quickly glance at our first five **actual observations** vs our first five **predicted observations**

```
actual_v_predicted = np.column_stack((y_test, predictions))
print(actual_v_predicted[0:5])
```

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

- Looks like our model did well at first glance!
- We will start next week with how to measure error with classification problems
- Then, we will optimize our model

Knowledge check 2

Exercise 2

Module completion checklist

Objective	Complete
Understanding classification and it's uses	✔
Summarize steps & application of kNN	✔
Clean and transform data to run kNN	✔
Define cross-validation and how and when it is used	✔
Implement kNN algorithm on the training data without cross-validation	✔
Identify performance metrics for classification algorithms and evaluate simple kNN model	
Find optimal hyperparameters with a grid search	
Use GridSearchCV to find optimal number of nearest neighbors and define optimal model	
Discuss reasons we would or would not use kNN	

Classification: assessing performance

- We have reviewed how to measure *error* when using **regression**
- Because our outcome variable is **binary**, we have a different way of measuring error than when the outcome variable was continuous

- The following terms are very important to measure performance of a classification algorithm
  - Confusion matrix
  - Accuracy
  - Receiver operating characteristic curve (ROC curve)
  - Area under the curve (AUC)

Classification: sklearn.metrics

- `sklearn.metrics` has many packages that are used to calculate metrics for various models
- We will be using metrics found within the *Classification metrics* section
- Here is an idea of what we can calculate using this library

Classification metrics

See the Classification metrics section of the user guide for further details.

`metrics.accuracy_score` (y_true, y_pred[, ...])	Accuracy classification score.
`metrics.auc` (x, y[, reorder])	Compute Area Under the Curve (AUC) using the trapezoidal rule
`metrics.average_precision_score` (y_true, y_score)	Compute average precision (AP) from prediction scores
`metrics.balanced_accuracy_score` (y_true, y_pred)	Compute the balanced accuracy
`metrics.brier_score_loss` (y_true, y_prob[, ...])	Compute the Brier score.
`metrics.classification_report` (y_true, y_pred)	Build a text report showing the main classification metrics
`metrics.cohen_kappa_score` (y1, y2[, labels, ...])	Cohen's kappa: a statistic that measures inter-annotator agreement.
`metrics.confusion_matrix` (y_true, y_pred[, ...])	Compute confusion matrix to evaluate the accuracy of a classification
`metrics.f1_score` (y_true, y_pred[, labels, ...])	Compute the F1 score, also known as balanced F-score or F-measure
`metrics.fbeta_score` (y_true, y_pred, beta[, ...])	Compute the F-beta score
`metrics.hamming_loss` (y_true, y_pred[, ...])	Compute the average Hamming loss.
`metrics.hinge_loss` (y_true, pred_decision[, ...])	Average hinge loss (non-regularized)
`metrics.jaccard_similarity_score` (y_true, y_pred)	Jaccard similarity coefficient score
`metrics.log_loss` (y_true, y_pred[, eps, ...])	Log loss, aka logistic loss or cross-entropy loss.
`metrics.matthews_corrcoef` (y_true, y_pred[, ...])	Compute the Matthews correlation coefficient (MCC)
`metrics.precision_recall_curve` (y_true, ...)	Compute precision-recall pairs for different probability thresholds
`metrics.precision_recall_fscore_support` (...)	Compute precision, recall, F-measure and support for each class
`metrics.precision_score` (y_true, y_pred[, ...])	Compute the precision
`metrics.recall_score` (y_true, y_pred[, ...])	Compute the recall
`metrics.roc_auc_score` (y_true, y_score[, ...])	Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores.
`metrics.roc_curve` (y_true, y_score[, ...])	Compute Receiver operating characteristic (ROC)
`metrics.zero_one_loss` (y_true, y_pred[, ...])	Zero-one classification loss.

Confusion matrix: what is it

- **Confusion matrix is what we use to measure error**
- We will use it to calculate Accuracy, Misclassification rate, True positive rate, False positive rate, and Specificity
- **In the matrix overview of our data, let `Y1` be "non-vulnerable" and `Y2` be "vulnerable"**

	Predicted Y1	Predicted Y2	Actual totals
Y1	True Negative (TN)	False Positive (FP)	Total negatives
Y2	False Negative (FN)	True Positive (TP)	Total positives
Predicted totals	Total predicted negatives	Total predicted positives	Total

Confusion matrix: accuracy

- We will now review the metrics we are looking for from the confusion matrix, one at a time

Accuracy: overall, how often is the classifier correct?

TP + TN / total

	Predicted Y1	**Predicted Y2**	**Actual totals**
Y1	**True Negative (TN)**	**False Positive (FP)**	Total negatives
Y2	**False Negative (FN)**	**True Positive (TP)**	Total positives
Predicted totals	Total predicted negatives	Total predicted positives	**Total**

Confusion matrix: misclassification rate

Misclassification rate (error rate) : overall, how often is the classifier wrong?

FP + FN / **total**

	Predicted Y1	**Predicted Y2**	**Actual totals**
Y1	True Negative (TN)	**False Positive (FP)**	Total negatives
Y2	**False Negative (FN)**	True Positive (TP)	Total positives
Predicted totals	Total predicted negatives	Total predicted positives	**Total**

Confusion matrix: true positive rate

True positive rate (Sensitivity): how often does it predict yes?

TP / **actual yes**

	Predicted Y1	**Predicted Y2**	**Actual totals**
Y1	True Negative (TN)	False Positive (FP)	Total negatives
Y2	False Negative (FN)	**True Positive (TP)**	**Total positives**
Predicted totals	Total predicted negatives	Total predicted positives	Total

Confusion matrix: false positive rate

False positive rate: when it's actually no, how often does it predict yes?

FP / **actual no**

	Predicted Y1	**Predicted Y2**	**Actual totals**
Y1	True Negative (TN)	**False Positive (FP)**	**Total negatives**
Y2	False Negative (FN)	True Positive (TP)	Total positives
Predicted totals	Total predicted negatives	Total predicted positives	Total

Confusion matrix: specificity

True Negative Rate (Specificity): when it's actually no, how often does it predict no?

TN / **actual no**

	Predicted Y1	**Predicted Y2**	**Actual totals**
Y1	**True Negative (TN)**	**False Positive (FP)**	**Total negatives**
Y2	**False Negative (FN)**	**True Positive (TP)**	Total positives
Predicted totals	Total predicted negatives	Total predicted positives	Total

Confusion matrix: summary

- Here is a table with all the metrics in one place:

Metric name	Formula
Accuracy	True positive + True Negative / Overall total
Misclassification rate	False positive + False Negative / Overall total
True positive rate	True positive / Actual yes (True positive + False negative)
False positive rate	False positive / Actual no (False positive + True negative)
Specificity	True negative / Actual no (False positive + True negative)

Confusion matrix in Python

- Now that we know the metrics behind the madness, let's execute the code to build a confusion matrix in Python that **we** can see
- We use a function `confusion_matrix` from `sklearn.metrics`

```
# Confusion matrix for knn.
cm_knn5 = confusion_matrix(y_test, predictions)
print(cm_knn5)
```

```
[[ 294  768]
 [ 366 1440]]
```

- We won't go through all of the metrics right now, but let's calculate accuracy because it's a metric used frequently to compare classification models

- **Accuracy = True positive + True Negative / Overall total**
- Using `accuracy_score` from `sklearn.metrics`, we calculate:

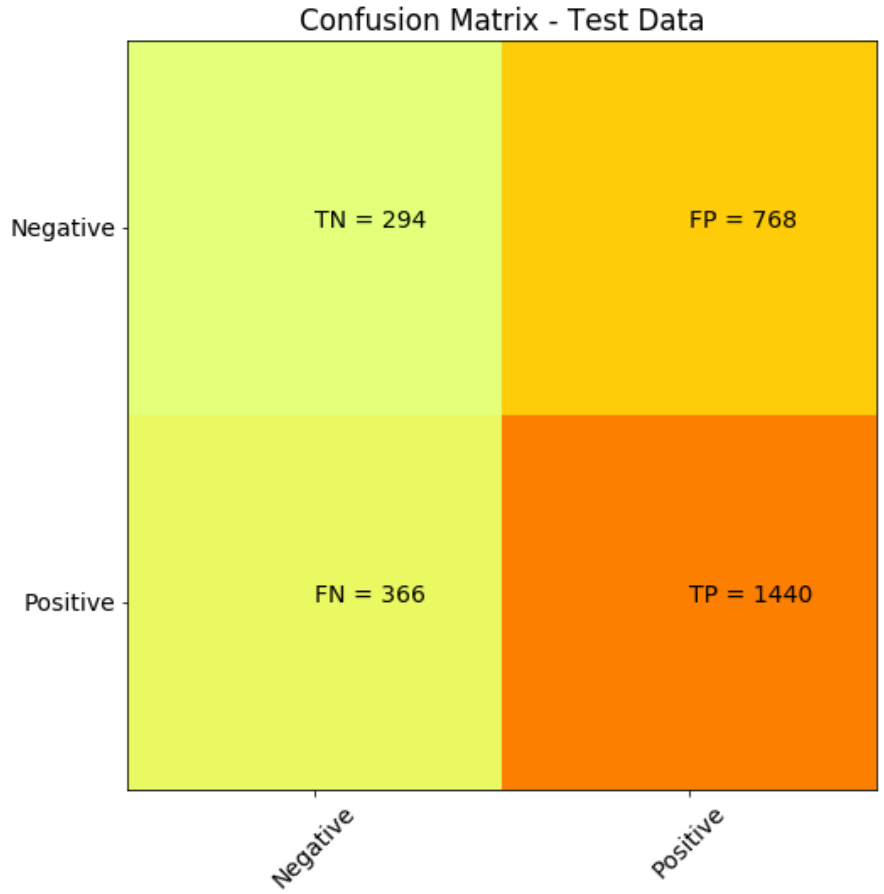
```
print(round(accuracy_score(y_test, predictions), 4))
```

```
0.6046
```

Confusion matrix: visualize

- Let's visualize our confusion matrix

```
plt.imshow(cm_knn5, interpolation =
'nearest', cmap = plt.cm.Wistia)
classNames = ['Negative', 'Positive']
plt.title('Confusion Matrix - Test
Data')
plt.ylabel('True label')
plt.xlabel('Predicted label')
tick_marks =
np.arange(len(classNames))
plt.xticks(tick_marks, classNames,
rotation = 45)
plt.yticks(tick_marks, classNames)
s = [['TN', 'FP'], ['FN', 'TP']]
for i in range(2):
    for j in range(2):
        plt.text(j,i, str(s[i][j]) + "
= " + str(cm_knn5[i][j]))
plt.show()
```



Evaluation of kNN with 5 neighbors

- Let's store the accuracy of this model:

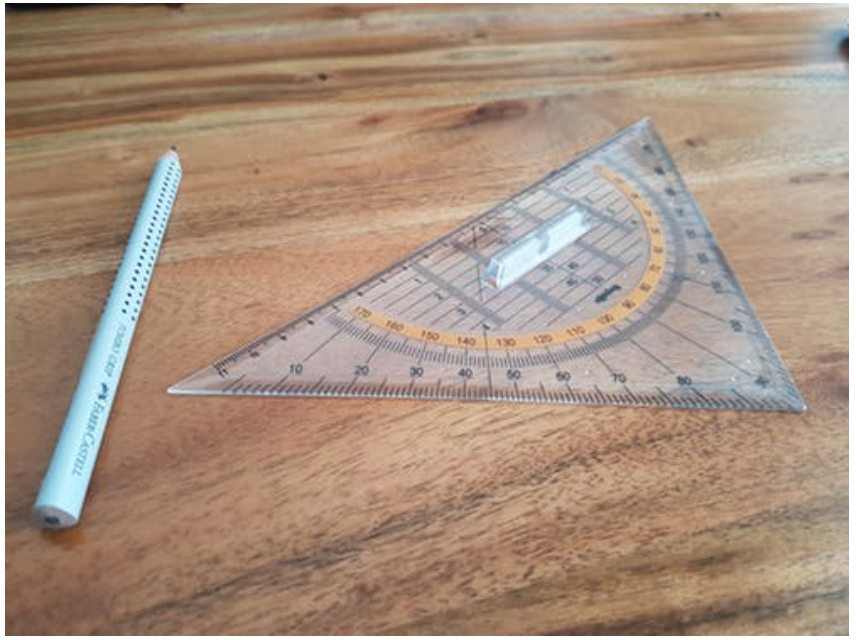
```
# Create a dictionary with accuracy values for our knn model with k = 5.
model_final_dict = {'metrics': ["accuracy"],
                    'values':[round(accuracy_score(y_test, predictions), 4)],
                     'model':['knn_5']}
model_final = pd.DataFrame(data = model_final_dict)
print(model_final)
```

```
    metrics  values  model
0  accuracy  0.6046  knn_5
```

- Our model is not doing great, but we will now observe how it does over the next week
- **Next, we are going to try and optimize it even more by using cross-validation**
- As we move forward, we will add our accuracy scores for each new model we build
- **This way, we can compare all classification models and evaluate which one seems to be the *model champion***

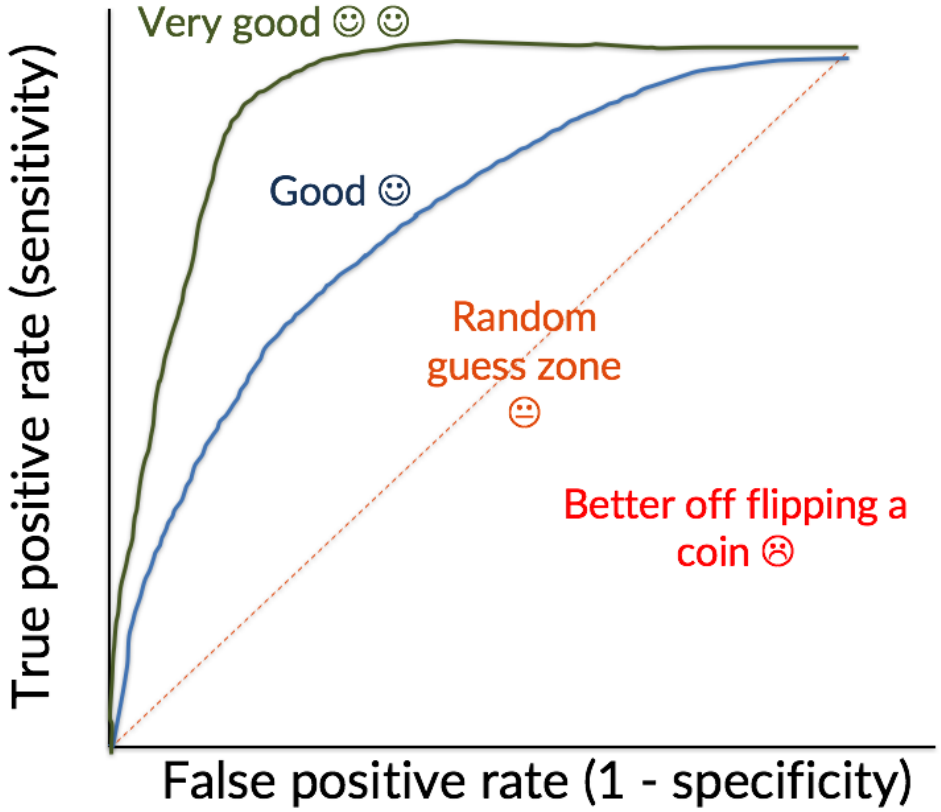
Performance of our kNN model

- The remaining metrics we want to look at to evaluate our model are:
  - **Receiver operating characteristic curve (ROC curve)**
  - **Area under the curve (AUC)**



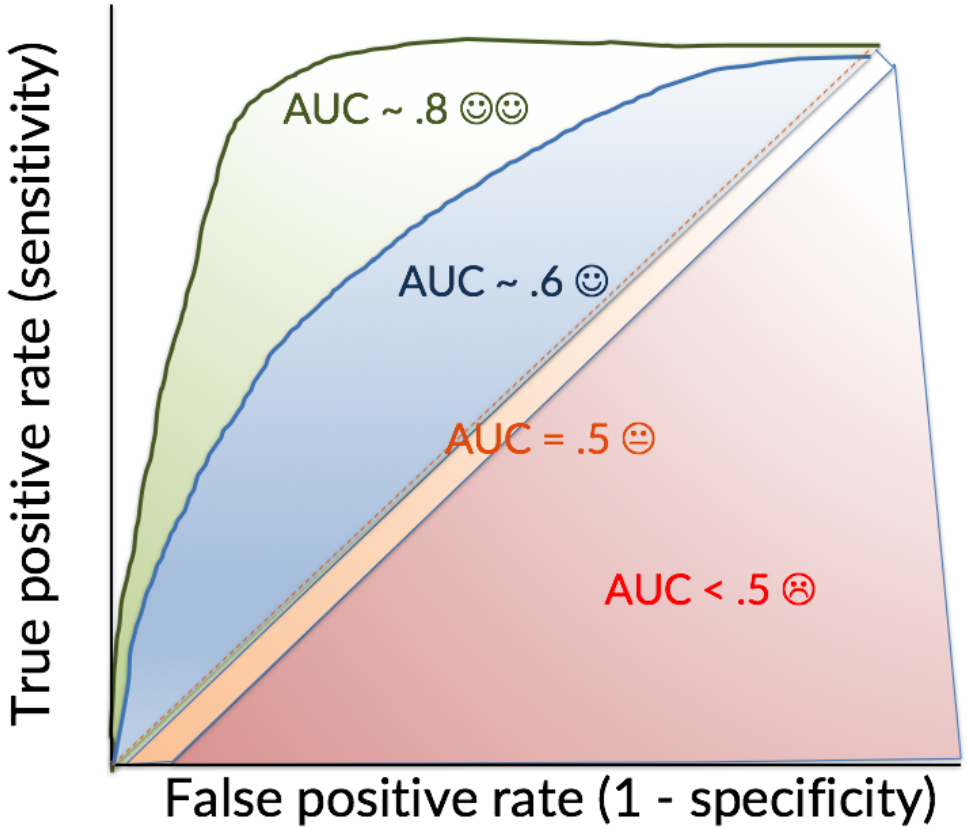
ROC: receiver operator characteristic

- It is a plot of the **true positive rate (TPR)** against the **false positive rate (FPR)**
- The plot illustrates the tradeoff between **TPR** and **FPR**
- Classification models produce them to show the performance of the model and allow us to choose which threshold to use



AUC: area under the 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 perfect **AUC** = 1 (you will never see this number working with real world data!)



Plot ROC and calculate AUC

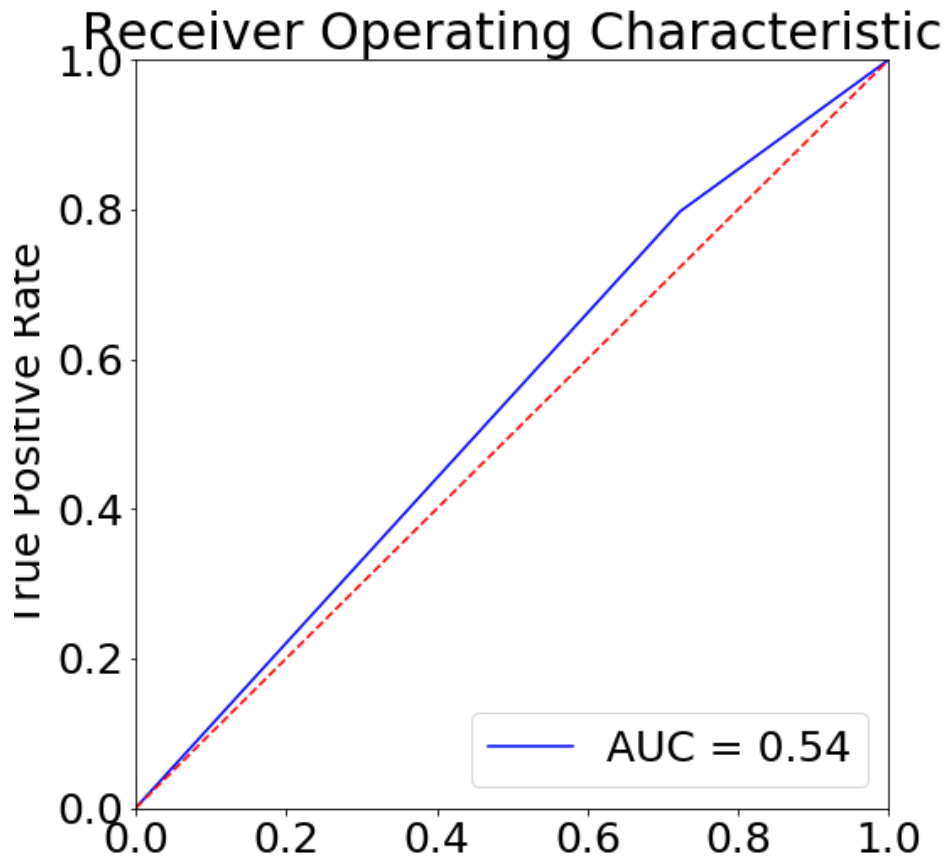
- Let's plot the **ROC** for our model and calculate the **AUC**

```
# Store FPR, TPR, and threshold as variables.
fpr, tpr, threshold = metrics.roc_curve(y_test,
predictions)
# Store the AUC.
roc_auc = metrics.auc(fpr, tpr)
```

```
plt.title('Receiver Operating Characteristic')
plt.plot(fpr, tpr, 'b', label = 'AUC = %0.2f' %
roc_auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```

```
(0, 1)
```

```
(0, 1)
```



Knowledge check 3

Exercise 3

Module completion checklist

Objective	Complete
Understanding classification and it's uses	✔
Summarize steps & application of kNN	✔
Clean and transform data to run kNN	✔
Define cross-validation and how and when it is used	✔
Implement kNN algorithm on the training data without cross-validation	✔
Identify performance metrics for classification algorithms and evaluate simple kNN model	✔
Find optimal hyperparameters with a grid search	
Use GridSearchCV to find optimal number of nearest neighbors and define optimal model	
Discuss reasons we would or would not use kNN	

Finding optimal k

1. **We have now:**
    i. Ran kNN on our training data
    ii. Predicted using the kNN model on our test data
    iii. Reviewed performance metrics for classification algorithms
    iv. Built a confusion matrix for the predicted model
2. **We will now:**
    i. Find what our optimal k value is
    ii. Evaluate the new predictions

Parameters vs. hyperparameters

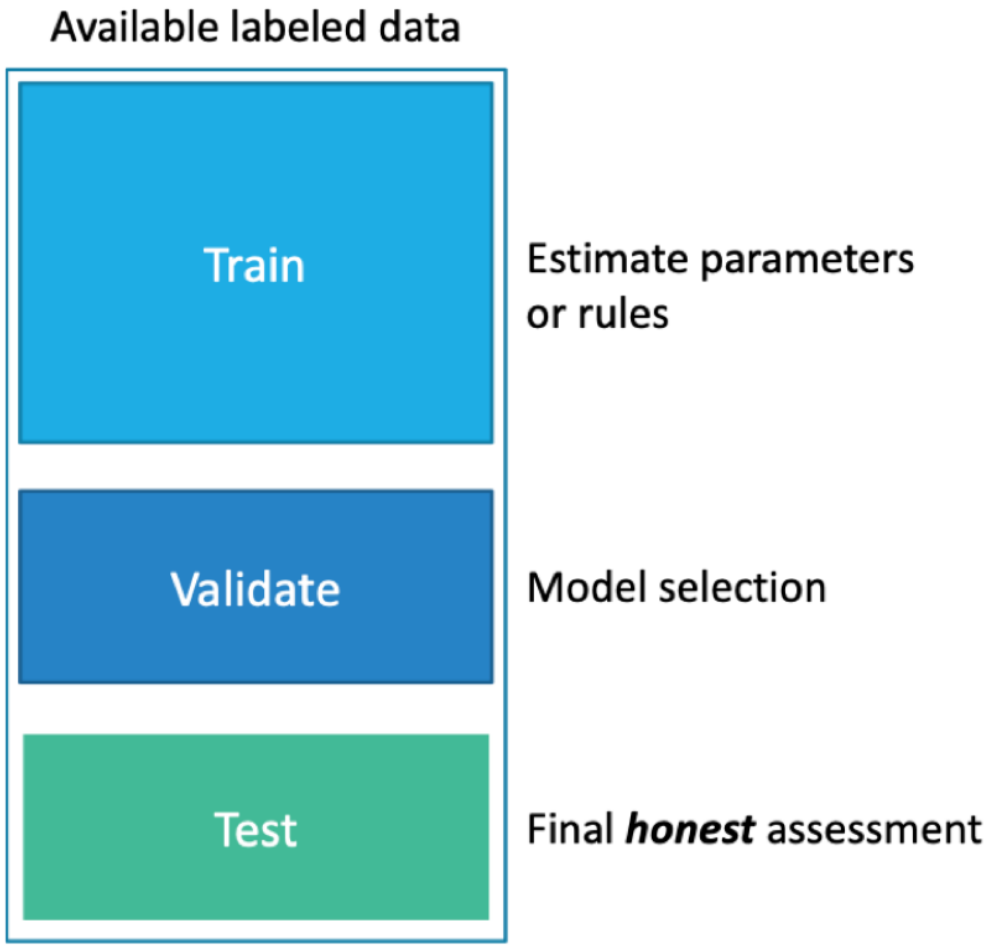
- **Parameters**
  - **Parameters** are derived from training data
  - Example: the weights of a predictor in a regression
  - They are learned by the algorithm from the data
- **Hyperparameters**
  - **Hyperparameters** are manually set before the training process
  - Example: k in kNN, number of trees in random forest, penalty in penalized regression
  - They are found using a **grid search**

What is a grid search?

- **Grid search helps us find the optimal hyperparameters**
  - Grid search allows us to **search over a list of hyperparameter values** to find the optimal hyperparameter
  - It is a brute force approach: it **creates a model for every hyperparameter value** in the list
  - We then choose the hyperparameter value that **created the model with the smallest error**
- **Example kNN**
  - Grid search allows us to find the optimal `k`
  - We can use a grid search to search over k=1, k=2, k=3, k=4, etc.
  - Grid search creates models for every value of `k` in the list
  - We can choose the `k` that yields the smallest error

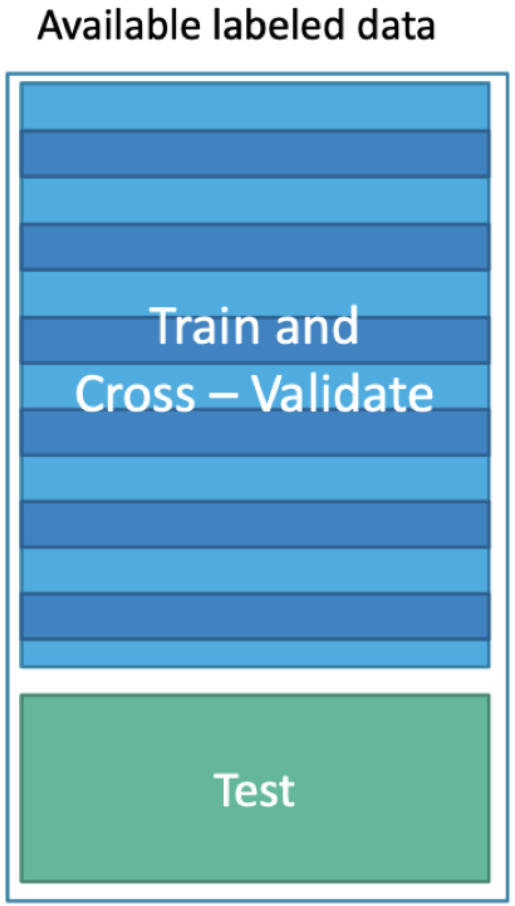
Finding the smallest error with grid search

- So far, we have used a **train - test split** to train and evaluate our models
- This worked because we only had parameters, but no hyperparameters
- Now that we **also have hyperparameters in our models, we need a train - validation - test split**
- The validation set allows us to compare the different models created by the grid search and **choose the optimal hyperparameters**



Using cross validation with grid search

- Instead of using train - validation - test split, we can use **cross-validation**
- Cross-validation allows us to **perform the split multiple times** on the same dataset
- We have new **train and validation sets** for each fold `n`
- This leads to **more accurate results**, but is computationally intensive
- It is best suited for **small datasets**
- Today, we will use cross validation with our grid search



- Estimate parameters or rules
- Model selection and hyper-parameter tuning

Final ***honest*** assessment

Keeping test data separate

True assessment

- No matter if you use train - validation - test or cross-validation, you always want to have a **separate test set**
- The test set **must not be involved in the model training and selection process**
- This allows for a **true assessment**

Why this is important

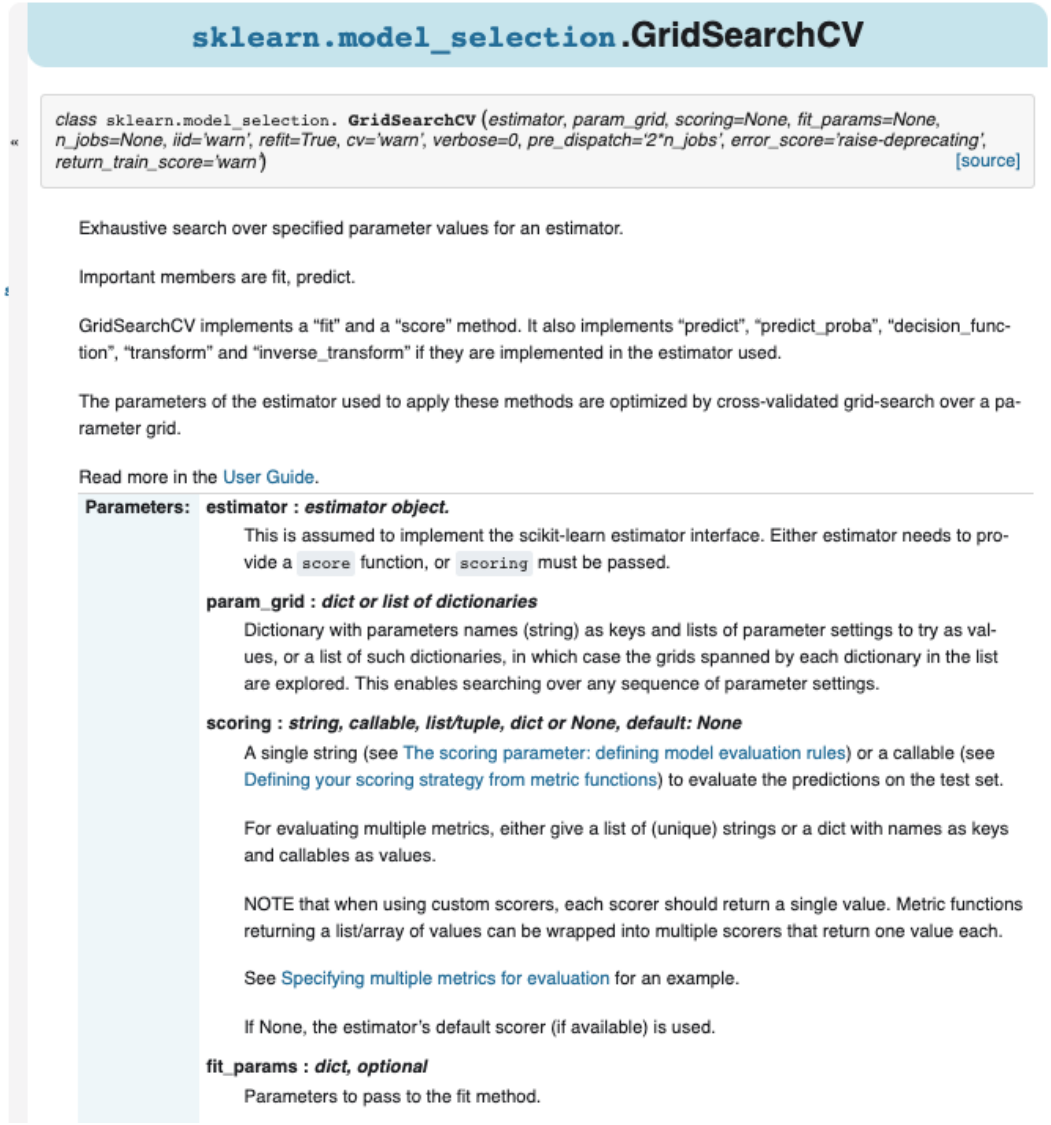
- Only a honest assessment tells you how your model will perform on **previously unseen data**
- The **errors on your train and validation sets will be lower** than the error you can expect on previously unseen data

Module completion checklist

Objective	Complete
Understanding classification and it's uses	✔
Summarize steps & application of kNN	✔
Clean and transform data to run kNN	✔
Define cross-validation and how and when it is used	✔
Implement kNN algorithm on the training data without cross-validation	✔
Identify performance metrics for classification algorithms and evaluate simple kNN model	✔
Find optimal hyperparameters with a grid search	✔
Use GridSearchCV to find optimal number of nearest neighbors and define optimal model	
Discuss reasons we would or would not use kNN	

GridSearchCV for optimal hyperparameters

- Once again, `GridSearchCV` is different from cross-validation because it performs an exhaustive search over a range of specified hyperparameter values
- Let's now learn a little more about `GridSearchCV`



sklearn.model_selection.GridSearchCV

class sklearn.model_selection. **GridSearchCV** (*estimator, param_grid, scoring=None, fit_params=None, n_jobs=None, iid='warn', refit=True, cv='warn', verbose=0, pre_dispatch='2*n_jobs', error_score='raise-deprecating', 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.

Read more in the User Guide.

Parameters:

estimator : *estimator object.*
This is assumed to implement the scikit-learn estimator interface. Either estimator needs to provide a `score` function, or `scoring` must be passed.

param_grid : *dict or list of dictionaries*
Dictionary with parameters names (string) as keys and lists of parameter settings to try as values, or a list of such dictionaries, in which case the grids spanned by each dictionary in the list are explored. This enables searching over any sequence of parameter settings.

scoring : *string, callable, list/tuple, dict or None, default: None*
A single string (see The scoring parameter: defining model evaluation rules) or a callable (see Defining your scoring strategy from metric functions) to evaluate the predictions on the test set.

For evaluating multiple metrics, either give a list of (unique) strings or a dict with names as keys and callables as values.

NOTE that when using custom scorers, each scorer should return a single value. Metric functions returning a list/array of values can be wrapped into multiple scorers that return one value each.

See Specifying multiple metrics for evaluation for an example.

If None, the estimator's default scorer (if available) is used.

fit_params : *dict, optional*
Parameters to pass to the fit method.

Finding optimal k

- We will use the `GridSearchCV` function now so that we can find the optimal number of `k`
- As you walk through the steps, keep in mind that this is a method that we can use to tune various hyperparameters depending on the model
- Let's look at the steps below:

1. Define the hyperparameter value (in this case, we are defining the range of `k`)
2. Create a hyperparameter grid that maps the values that to be searched
3. Fit the grid with data from the model
4. View the scores and choose the optimal hyperparameter
5. View the hyperparameters of the "best model"

Finding optimal k - GridSearchCV

```
# Define the parameter values that should be searched.
k_range = list(range(1, 31))

# Create a parameter grid: map the parameter names to the values that should be searched by building a
Python dictionary.
# key: parameter name
# value: list of values that should be searched for that parameter
# single key-value pair for param_grid
param_grid = dict(n_neighbors = k_range)
print(param_grid)

# Instantiate the grid using our original model - knn with k = 3.
```

```
{'n_neighbors': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,
25, 26, 27, 28, 29, 30]}
```

```
grid = GridSearchCV(knn, param_grid, cv = 10, scoring = 'accuracy')
```

Finding optimal k - GridSearchCV

```
# Fit the grid with data.
grid.fit(X_scaled, y)
```

```
GridSearchCV(cv=10, error_score='raise-deprecating',
             estimator=KNeighborsClassifier(algorithm='auto', leaf_size=30,
                                            metric='minkowski',
                                            metric_params=None, n_jobs=None,
                                            n_neighbors=5, p=2,
                                            weights='uniform'),
             iid='warn', n_jobs=None,
             param_grid={'n_neighbors': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,
                                         13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
                                         23, 24, 25, 26, 27, 28, 29, 30]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=False,
             scoring='accuracy', verbose=0)
```

Finding optimal k - view results

```
# View the complete results (list of named tuples).
print(grid.cv_results_['mean_test_score'])
```

```
[0.56293816 0.53311709 0.58198179 0.5597991  0.57967982 0.56816993
 0.58417914 0.57120435 0.57926127 0.57915664 0.59527048 0.58784137
 0.58627184 0.58009836 0.59788637 0.58187716 0.59684001 0.589097
 0.59715392 0.59684001 0.6015486  0.5944334  0.59485194 0.5908758
 0.60876844 0.60416449 0.6133724  0.6136863  0.61881344 0.61347703]
```

Finding optimal k

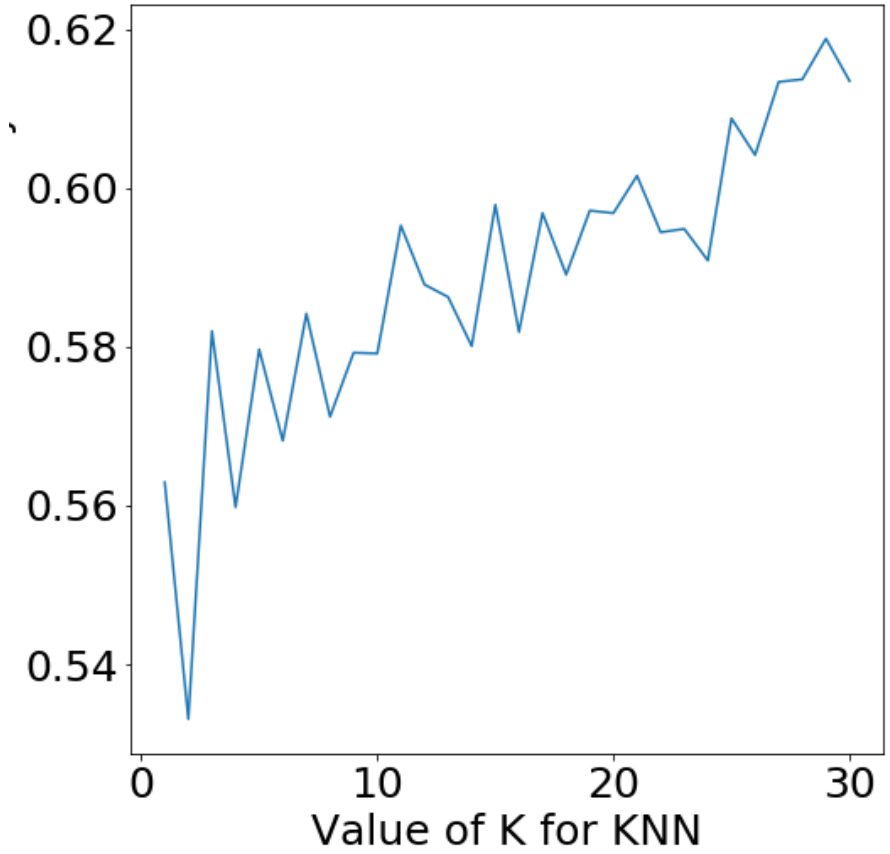
- First, we create a list of mean scores

```
# Create a list of the mean scores only by using a list comprehension to loop through grid.cv_results_.
grid_mean_scores = [result for result in grid.cv_results_['mean_test_score']]
print(grid_mean_scores)
```

```
[0.5629381605106205, 0.5331170869519724, 0.5819817934498274, 0.559799100136026, 0.5796798158417914,
0.5681699278016114, 0.5841791357120435, 0.5712043528303861, 0.579261274458512, 0.5791566391126922,
0.5952704823689442, 0.5878413728157371, 0.5862718426284399, 0.5800983572250706, 0.5978863660144397,
0.5818771581040075, 0.5968400125562415, 0.5890969969655749, 0.5971539185937009, 0.5968400125562415,
0.6015486031181333, 0.5944333996023857, 0.594851940985665, 0.5908757978445118, 0.6087684419797007,
0.6041644867636288, 0.6133723971957727, 0.6136863032332321, 0.6188134351784033, 0.6134770325415926]
```

Finding optimal k - plot

```
# Plot the results.
plt.plot(k_range, grid_mean_scores)
plt.xlabel('Value of K for KNN')
plt.ylabel('Cross-Validated Accuracy')
```



Define and examine the optimized model

- Now that we have found our optimal `k`, let's examine our results:
  - best accuracy score over all parameters of `k`
  - parameters that led to that score
  - model that was run to achieve that score

```
# Single best score achieved across all params (k).
print(grid.best_score_)
```

```
0.6188134351784033
```

```
grid_score = grid.best_score_

# Dictionary containing the parameters (k) used to generate
that score.
print(grid.best_params_)

# Actual model object fit with those best parameters.
# Shows default parameters that we did not specify.
```

```
{'n_neighbors': 29}
```

```
print(grid.best_estimator_)
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30,
metric='minkowski',
                     metric_params=None, n_jobs=None,
n_neighbors=29, p=2,
                     weights='uniform')
```

Add GridSearchCV score to the final scores

- So we have it, let's add this score to the dataframe we created earlier

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

```
    metrics  values              model
0  accuracy  0.6046              knn_5
1  accuracy  0.6188  knn_GridSearchCV
```

Optimal model and final thoughts

- We can use the model we just built to predict on the test set
- It is optimized because of the cross-validation and will use the optimized k, which is `k = 29`
- Although it is an optimized model, it does not mean it will perform better than our previous model, it is just more accurate
- This is essential when finding a model champion

```
knn_best = grid.best_estimator_

# Check accuracy of our model on the test data.
print(knn_best.score(X_test, y_test))
```

```
0.6286610878661087
```

```
knn_champ = knn_best.score(X_test, y_test)
```

Model champion dataframe

- Let's add our final model to the dataframe 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!

```
# Add this final model champion to our dataframe.
model_final = model_final.append({'metrics' : "accuracy" ,
                                  'values' : round(knn_champ, 4),
                                  'model':'knn_29' } ,
                                  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
```

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

Module completion checklist

Objective	Complete
Understanding classification and it's uses	✔
Summarize steps & application of kNN	✔
Clean and transform data to run kNN	✔
Define cross-validation and how and when it is used	✔
Implement kNN algorithm on the training data without cross-validation	✔
Identify performance metrics for classification algorithms and evaluate simple kNN model	✔
Find optimal hyperparameters with a grid search	✔
Use GridSearchCV to find optimal number of nearest neighbors and define optimal model	✔
Discuss reasons we would or would not use kNN	

KNN pros and cons

Pros

- Easy to use
- Can easily handle multiple categories
- There are many options to adjust (which features to use, measurement metric, etc)

Cons

- **There are many options to adjust**
- The correct distance metric is important
- Can be slow with large amounts of data
- **Gets less accurate with more predictors, will not be accurate on larger datasets**
- It is **LAZY** - it memorizes and uses every data point when calculating kNN

Knowledge check 4

Exercise 4

Module completion checklist

Objective	Complete
Understanding classification and it's uses	✔
Summarize steps & application of kNN	✔
Clean and transform data to run kNN	✔
Define cross-validation and how and when it is used	✔
Implement kNN algorithm on the training data without cross-validation	✔
Identify performance metrics for classification algorithms and evaluate simple kNN model	✔
Find optimal hyperparameters with a grid search	✔
Use GridSearchCV to find optimal number of nearest neighbors and define optimal model	✔
Discuss reasons we would or would not use kNN	✔

Workshop: Next steps!

- 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
- This is an exploratory exercise to get you comfortable with the content we discussed today

Today you will

- Run a simple kNN model
- Use performance metrics covered in class to assess the model
- Evaluate the optimal number of nearest neighbors using GridSearchCV and build the optimal model
- Save accuracy metrics of all the models to the `model_final_workshop` dataframe

This completes our module

Congratulations!