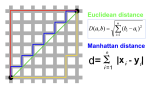
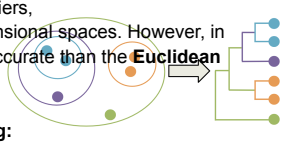
Ch 7 - Clustering

- Goal of Clustering: Assign labels to groups that do not already have reliable labels assigned to them, based on data.
- Distance Measures: "Central to...cluster analysis is the notion of the degree of similarity (or dissimilarity) between the individual objects being clustered. A clustering method attempts to group the objects based on the definition of similarity supplied to it."



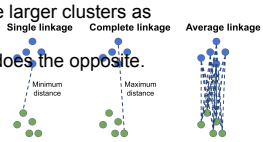
- The **Manhattan distance** may be less sensitive to outliers, especially in high-dimensional spaces. However, in other cases, it is less accurate than the **Euclidean distance**.



- 2 main types:

Hierarchical clustering:

- In **agglomerative** hierarchical clustering, we start with individual samples and create larger clusters as we go.
- **Divisive** hierarchical clustering does the opposite.



Overall Distance →

Partition-based clustering:

ex) **k-means clustering** →

- One characteristic (and limitation) of k-means clustering is that we must pre-specify a value for k.

Disadvantages:

- It's hard to know how many clusters to define.
- Boundaries between clusters may be ambiguous.
- Algorithms may not deal well with noisy data.

Binary Classification tools:

Keras

- Created by François Chollet in 2015.
- Written for Python.
- Initially it used Theano (now discontinued) as its "back end."
- TensorFlow became the official back end in 2017.

TensorFlow

- Created by Google Brain in 2015.
- Optimized for performing complex mathematical operations, including gradient descent.
- It can operate on different types of hardware—using the same code.

Using keras for binary classification:

Step 1: Separate into training, validation, and test sets.

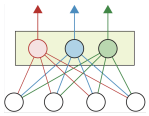
Step 2: Separate features and labels and convert to a numeric matrix (X) and a vector (y)

Step 3:

- Input dimensions = # of columns in X.
- Output dimensions:
  - 1 for binary classification.
  - # of classes for multiclass classification.

Step 4: Design the model.

A fully connected ("dense") layer →



Step 5: Implement the model.

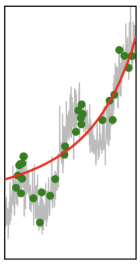
```
model = keras_model_sequential() %>%
  layer_dense(units = 64,
              activation = "relu",
              input_shape = input_dim) %>%
  layer_dense(units = 64,
              activation = "relu") %>%
  layer_dense(units = output_dim,
              activation = "sigmoid")
```

Step 6: Compile the model.

```
compile(model, optimizer = "rmsprop",
  loss = "binary_crossentropy",
  metrics = "accuracy")
```

Step 7: Fit the model.

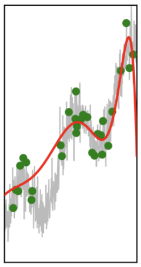
```
history = model %>% fit(
  x = X_train,
  y = y_train,
  epochs = 50,
  batch_size = 32,
  validation_data = list(X_val, y_val))
```



Step 8: Evaluate the model.

```
evaluation = evaluate(model,
                      X_test,
                      y_test)
print(evaluation)

loss: 0.6857 - accuracy: 0.5455
```



Step 9: Make predictions for test set.

```
predictions = predict(model, X_test)
print(head(predictions, n = 4))

           [,1]
[1,] 0.4410261
[2,] 0.4570570
[3,] 0.4639179
[4,] 0.4628421
```

Ch 8 - Training & Testing

- name vector elements names() = c() (unname too)
- rbind() cbind() - bind rows or cols in df
- slice_sample(data, n=2) (select 2 random rows)
- slice_sample(data, prop=1) shuffle all rows

Using keras for multiclass classification:

Step 1 - 9 same as binary with slight differences:

- step 5 "softmax" rather than "sigmoid"

The **softmax** function in Keras allows a neural network to handle multiple classes simultaneously. It assigns probabilities across all the classes in a single step, meaning it does not explicitly use a one-versus-all or one-versus-one strategy.

- step 6 "categorical_crossentropy" rather than "binary_crossentropy"

Multilabel classification is a variation on multiclass classification in which labels are not mutually exclusive. Each sample can be assigned multiple labels with high probability.

The probabilistic predictions change in nature. With **multiclass classification**, the probabilities for each sample sum to one.

With **multilabel classification**, a sample can have multiple labels with high probabilities. Alternatively, a sample can have low probabilities for all labels.

Hyperparameter: A configuration setting that controls how a model learns (e.g., number of clusters, learning rate, number of layers).

When to use training and test sets (only):

- Your dataset is small.
- You're committed to 1 particular algorithm
- Your algorithm is relatively simple and thus does not require tuning.

A validation set is useful for:

- Refining neural-network models across multiple training epochs.
- Selecting classification algorithm(s) automatically.
- Performing hyperparameter tuning.

In keras we can either explicitly give the model our validation data(done in ex), or have keras select it: history = model %>% fit(… validation_split = 0.25)

Cross validation: Repeatedly performing the training and testing process to get a rough accuracy estimate

- "Monte carlo" random test set each time

- k-fold

- nested k-fold - k fold within the training data

Ch 9 - Overfitting and Underfitting

```
select(data, colName)
select(data, everything())
select(data, starts_with("gr"))
select(data, all_of(c("weight", "group")))
Data = dplyr:: rename(data, NewName = OldName)
Tibble → matrix-converts all values to same dataType
Data = as.matrix(data)
```

Retrieving one column from a matrix:

data2 = data[,2]

Keeping as a matrix:

```
data3 = data[,2,drop=FALSE]
```

View documentation of a function:

```
?loss_kullback_leibler_divergence
```

ifelse() with mutate()

```
mutate(PlantGrowth,
  heavy = ifelse(weight > 5, "yes", "no"))
```

Viewing all metric functions:

```
ls(pattern = "^metric_", "package:keras")
```

Using/Customizing these functions:

```
compile(model,
        optimizer = optimizer_rmsprop(learning_rate = 5),
        loss = loss_binary_crossentropy(label_smoothing = 0.5),
        metrics = list(
          metric_accuracy(),
          metric_auc(thresholds = 500L),
          metric_precision(),
          metric_recall()
        ))
```

← High Bias, Low variance

- The lines do not closely match the actual data. The models are simple and consistent but not very accurate (underfit).

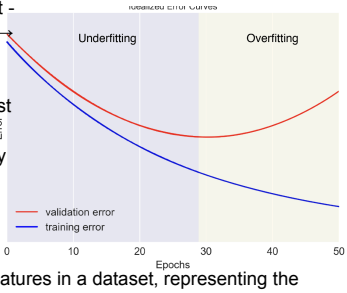
← Low Bias, High variance

- The lines match the actual data fairly well. The models are more complex. They are more accurate but less inconsistent (overfit).

Regularization techniques. The basic idea is to impose a "penalty" for having overly complex models, extremely large coefficients (weights), too many features, etc. This can encourage models that generalize better to unseen data.

ex) L1 (Lasso) - L2 (Ridge) - Elastic Net (L1L2 combo) - Dropout -

Early Stopping →



Data hygiene:

When we always make sure the test data are kept separate and only used once, after training is fully complete.

Dimensions:

The number of features in a dataset, representing the number of axes in the feature space.

Ch 10 - Data Preparation

Density: A measure of how sparsely data points are distributed across a feature space.

Weird: high-dimensional data operate in ways that are not intuitive to humans, needs complex math

Blessing of non-uniformity (or "blessing of structure"): A term that attempts to describe the reality that in high-dimensional spaces, classifiers can often identify meaningful patterns, despite "weirdness," because data do not spread uniformly in the sample space.

Convert column to a factor:

```
mutate(PlantGrowth, group = factor(group))
```

Calculating count per group:

```
group_by(PlantGrowth, group) %>%
  summarize(group_count = n())
```

Excluding rows with NA values in a given column:

```
data = mutate(data, filter(!is.na(MyColumn))
```

Step 0: Clean the data.

Step 1: Separate into training and test sets.

Step 2: Create the recipe.

Ex:

```
recp = recipe(Class ~ ., data =
training_data) %>%
  step_select(ColA, ColB, ColC) %>%
  step_impute_mode(ColA) %>%

step_impute_mean(all_numeric_predictors()
) %>%
  step_dummy(all_nominal_predictors()
one_hot = TRUE) %>%
  prep(training = training_data)
```

Step 3: "Bake" the training data.

Step 4: "Bake" the test data.

Step 5: Configure and fit the model.

Step 6: Make predictions.

Step 7: Combine predictions with class labels for the test set.

Step 8: Create confusion matrix (using discrete predictions).

Step 9: Calculate metrics for discrete predictions.

Step 10: Calculate area under ROC curve (for probabilistic predictions).

Step 11: Create ROC curve.

Enabling one-hot encoding:

step_dummy(..., one_hot = TRUE)

- "True" one-hot encoding refers to creating binary variables (also called dummy variables) for each category in a categorical feature, except for one category, which is implied as the baseline or reference. This is done to avoid the "dummy variable trap," when there is perfect multicollinearity—meaning that one of the dummy variables can be perfectly predicted using the others.
- "The goal of **principal component analysis** is to identify the most meaningful basis to re-express a data set. The hope is that this new basis will filter out the noise and reveal hidden structure."

Ch 11 - Classifiers

- **K - Nearest Neighbors**

non-parametric

Lazy learner - It doesn't build a model during the training phase and requires lots of memory. It only makes predictions when queried.

Distance-based - It relies on a distance metric such as the Euclidean or Manhattan distance. It assigns the class based on the majority class among the k closest neighbors.

Nonparametric - It does not make assumptions about the data's shape.

Sensitive to feature scaling: Normalization or standardization is typically required to ensure that features contribute equally to the distance metric.

Sensitive to k - A smaller k can lead to overfitting. A larger k might smooth out local patterns and lead to underfitting.

- **Decision Tree**

non-parametric

Hierarchical, rule-based structure - Each internal node represents a decision based on a feature, and each leaf node represents the final class label.

Nonparametric and interpretable - They do not make assumptions about the data's shape. They are highly interpretability—the model's decision process can be easily visualized.

Prone to overfitting - They can become overly complex and overfit the training data, particularly if a tree is too deep. Overfitting leads to poor generalization to unseen data. Techniques like pruning (removing unnecessary branches) or limiting the tree depth can mitigate this.

Sensitive to data variability - Decision Trees can be highly sensitive to small changes in the data. A slight change in the data might result in a completely different tree structure, leading to high variance.

- **Support Vector Machines (SVMs)**

parametric

Searches for optimal hyperplane - This hyperplane is chosen to maximize the margin (distance) between the nearest data points (called support vectors) from each class.

Effective for high-dimensional data - It often performs well when the number of features is larger than the number of data points, including when the samples are not linearly separable.

Uses a "kernel trick" - This trick transforms the input data into a higher-dimensional feature space where it then attempts to find a linear separator between the classes.

Uses regularization to avoid overfitting - The model complexity is controlled by tuning the "C" hyperparameter, which balances maximizing the margin and minimizing classification errors.

- **Naive Bayes**

parametric

Assumes a Gaussian distribution (is parametric) and feature independence: It assumes all features are independent of each other, given the class label. This simplifies the calculation of probabilities but may not hold true in real-world datasets.

Uses a probabilistic model: It calculates the posterior probability of each class, given the features.

Fast and efficient: It requires less computational power and time than many other classifiers and thus may be suitable for large datasets and real-time predictions.

Sensitive to zero probabilities: Naive Bayes can struggle when encountering feature values that do not appear in the training data, leading to zero probabilities.

Step 1: Assign folds.

Step 2: Create the recipe.

Step 3: Configure the model.

```
mod = rand_forest(trees = 10,
                  mode = "classification") %>%
  set_engine("ranger")
```

Step 4: Create a workflow.

Step 5: Train and test the models.

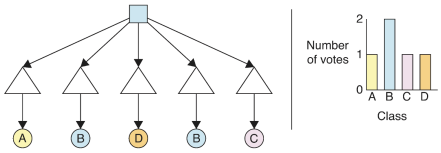
Step 6: Get the metrics.

Step 7: Get the predictions.

Ch 12 - Ensembles

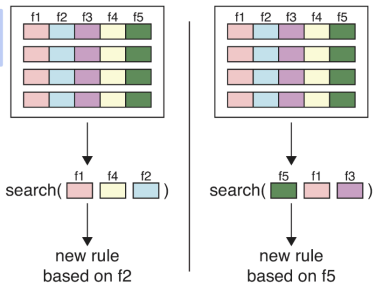
Bagging (bootstrap aggregation)

- Randomly select a bootstrap sample (with replacement).
- Make a prediction for each bootstrap sample.
- Use majority voting to determine the overall prediction.



Random Forest

- Randomly select a bootstrap sample.
- Randomly select a subset of features without replacement (feature bagging).
- Use Information Gain (or Gini Impurity) to decide how to split nodes (which of these features to use).
- Make a prediction for each sample.
- Use majority voting to determine the overall prediction.



Extra Trees (Extremely Randomized Trees)

- Randomly select a bootstrap sample (with replacement).
- Make a prediction for each bootstrap sample.

When doing so, it randomly chooses a split point for each feature, rather than using a metric like Information Gain to determine the split point.

This is an attempt to avoid overfitting (at a cost of reduced accuracy).

- Use majority voting to determine the overall prediction.

Boosting - Main ideas

Can be applied to any method of classification, but it is commonly applied to Decision Trees.

Combines many "weak" (simple, fast) classifiers.

Assigns a weight to each contributing classifier.

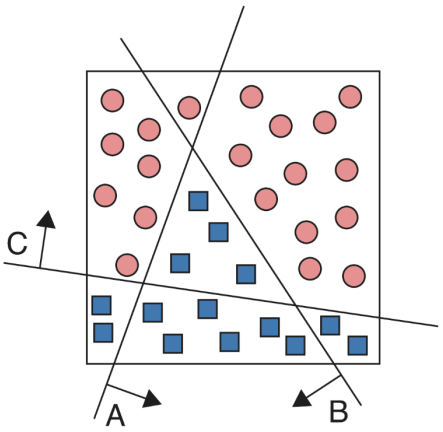
Boosting - Determining the weights

Each classifier's performance on the training data is its initial weight.

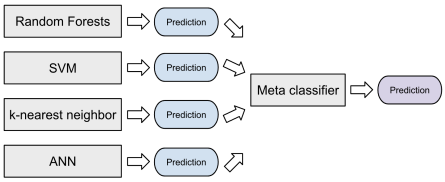
Weights are then assigned to each sample.

Misclassified samples receive the highest weights.

This process is repeated (and weights are adjusted) for multiple rounds, with the goal of improving the predictive performance for the samples that are most difficult to classify.



Multiple Classifier System



Stratified Sampling