K-nearest neighbors (KNN)

Joe Nese

Week 6, Class 1

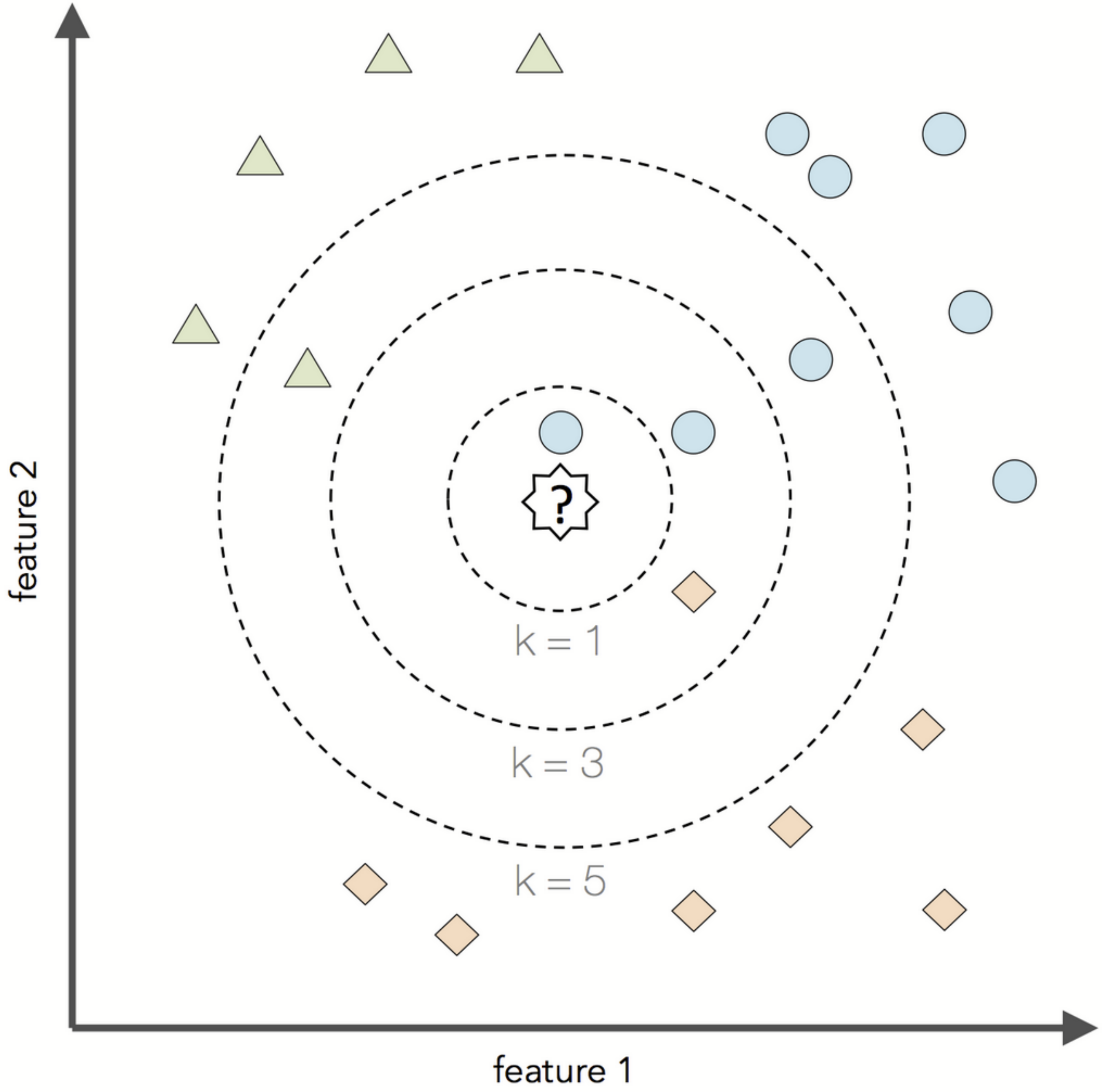
Agenda

- *K*-nearest neighbors model
  - regression
  - classification
  - imputation
- Non-regular grids
- Classification objective functions

K-nearest neighbors (*K*NN)

K-nearest neighbors (*K*NN)

- To predict the outcome of a new data point:
  - Finds the *K* most similar (nearest) data points in the predictor space
  - Take the average (regression) or mode (classification) outcome of those *K* cases
- A prediction is made using the training set outcomes for the neighbors (*K*)
- *K*NN stores the training set data and, when predicting new samples, locates the *K* training set points that are in the closest proximity to the new sample



This work by Sebastian Raschka is licensed under a
Creative Commons Attribution 4.0 International License.

K-nearest neighbors (*K*NN)

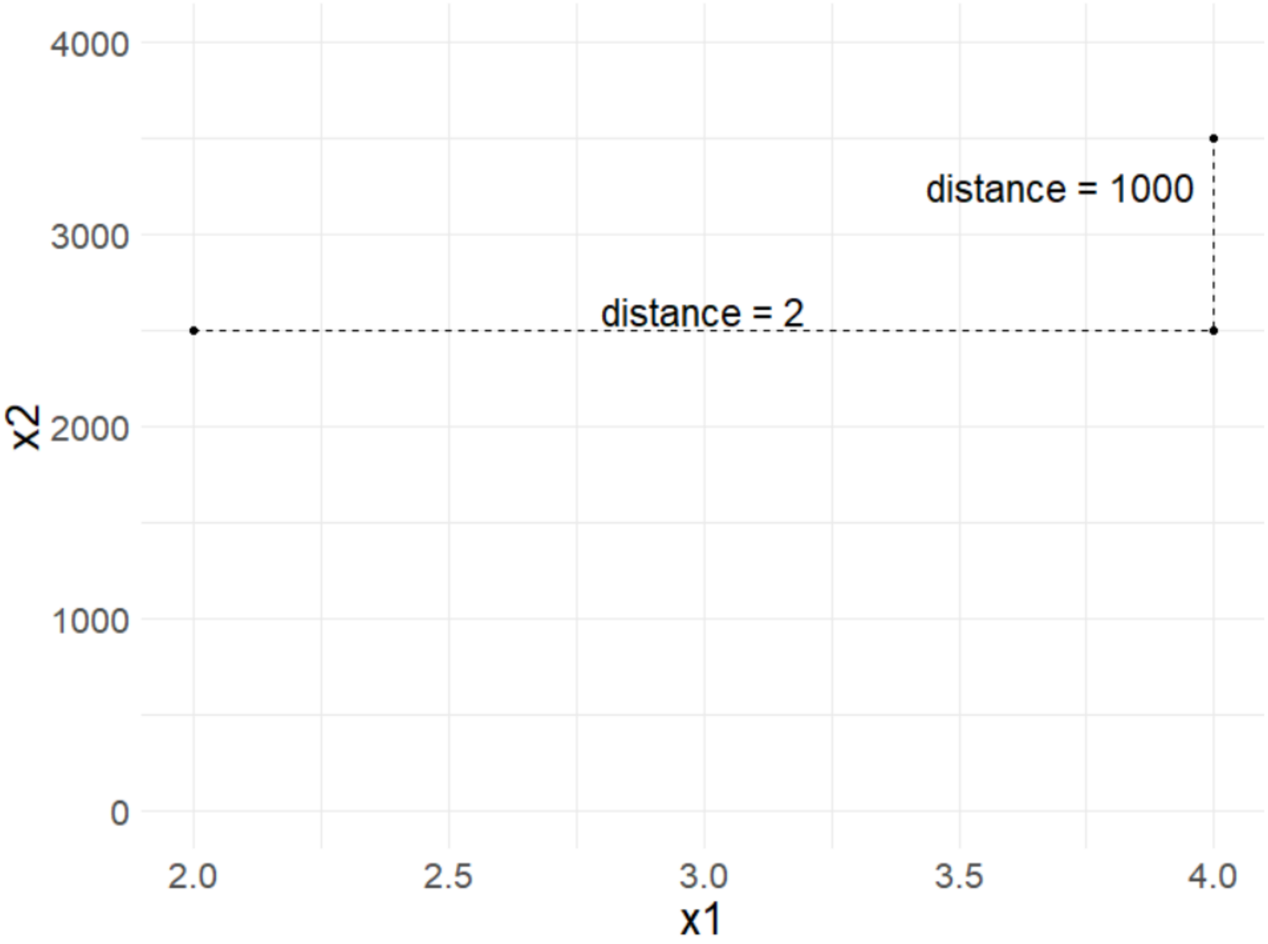
- *K*NN is a nonparametric method
  - Unlike parametric models, nonparametric models:
    - cannot be described by a fixed number of parameters that are being adjusted to the training set
    - the model structure is set *a priori* (and not defined by the training data)
    - do not assume that the data follow certain probability distributions (except Bayesian nonparametric methods)
    - make fewer assumptions about the data (than parametric methods)
- *K*NN uses lazy learning (or instance-based learning)
  - There is no training or model fitting stage
    - A *K*NN model literally stores the training data and uses it only at prediction time
    - Thus, each training instance represents a parameter in *K*NN model
    - Computationally inefficient

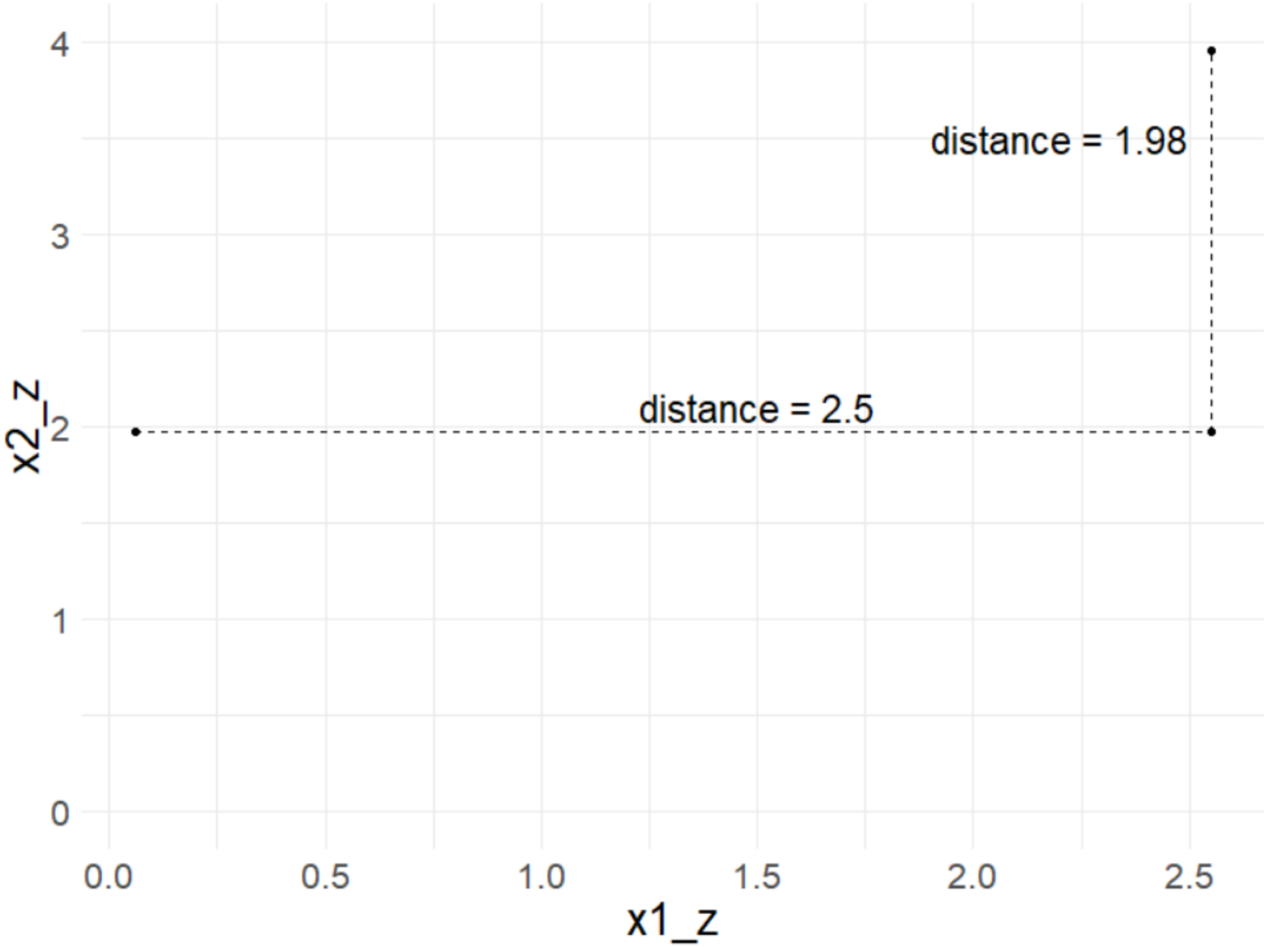
K-nearest neighbors (*K*NN)

- Feasible when the data contains more predictors than observations
- Requires the predictors to be in common units because the distance between predictors are used directly

  (like ridge, lasso models, elastic net, and support vector machines)

Scaling predictors





nearest_neighbor()

- nearest_neighbor()
  - {parsnip} model
- set_engine(“knn”)
  - {kknn} is the only engine for *K*NN in {tidymodels}
- the mode can be either regression or classification
  - set_mode("regression")
  - set_mode(“classification")

```
nearest_neighbor() %>%
  set_engine(“kknn") %>%
  set_mode(“classification")
```

`nearest_neighbor()` tuning parameters

`?nearest_neighbor`

```
nearest_neighbor(mode = "unknown", neighbors = NULL,
  weight_func = NULL, dist_power = NULL)
```

- `neighbors`: number of neighbors considered at each prediction
- `weight_func`: type of kernel function that weights the distances between samples
- `dist_power`: The parameter used when calculating the Minkowski distance
  - Manhattan distance with `dist_power` = 1
  - Euclidean distance with `dist_power` = 2

`defaults()`

“If left to their defaults here (NULL), the values are taken from the underlying model functions” from `{kknn}`

`neighbors` = 5

`weight_func` = “optimal”

`dist_power` = 2 (Euclidian)

neighbors

- The value of *K* controls the bias-variance
- With a small *K*, there is a potential for overfitting
  - imagine *K* = 1 would be very susceptible to changes in the data
  - low bias and high variance
  - smaller values of *K* tend to work best for high signal data with very few noisy (irrelevant) predictors
- With a large *K*, there is a potential to underfit
  - too many potentially irrelevant data points are used for prediction
  - high bias and lower variance
  - larger values of *K* tend to work best for data with more noisy (irrelevant) predictors in order to smooth out the noise

How do we find the most similar (nearest) neighbors?

- Two common measures of distance
  - Euclidian (as the crow flies)
  - Manhattan (city blocks)

How do we find the most similar (nearest) neighbors?

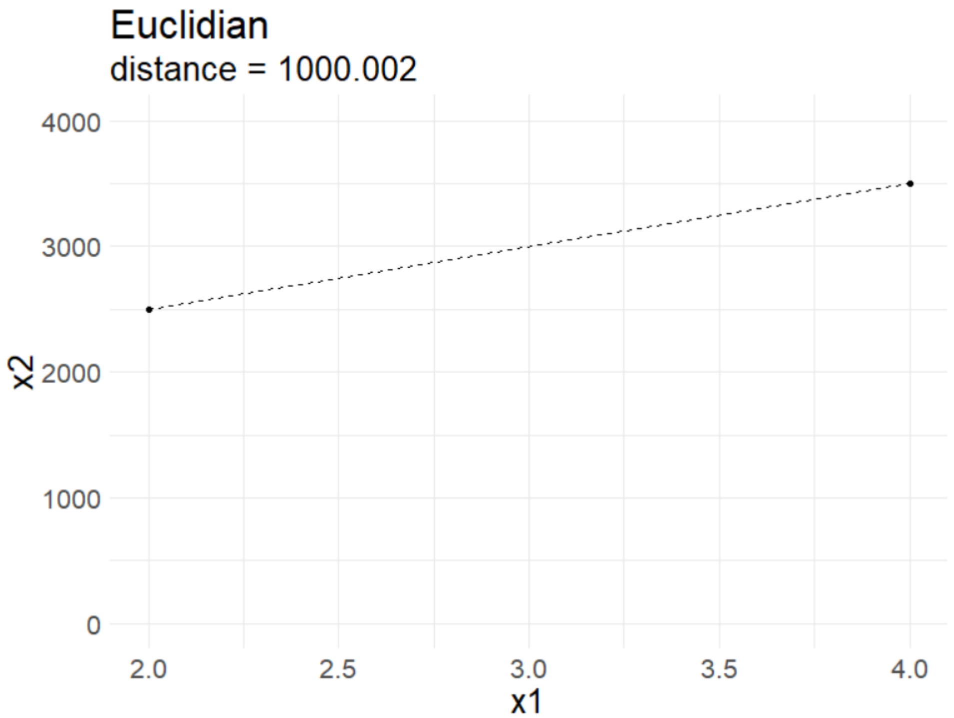
- Two common measures of distance:

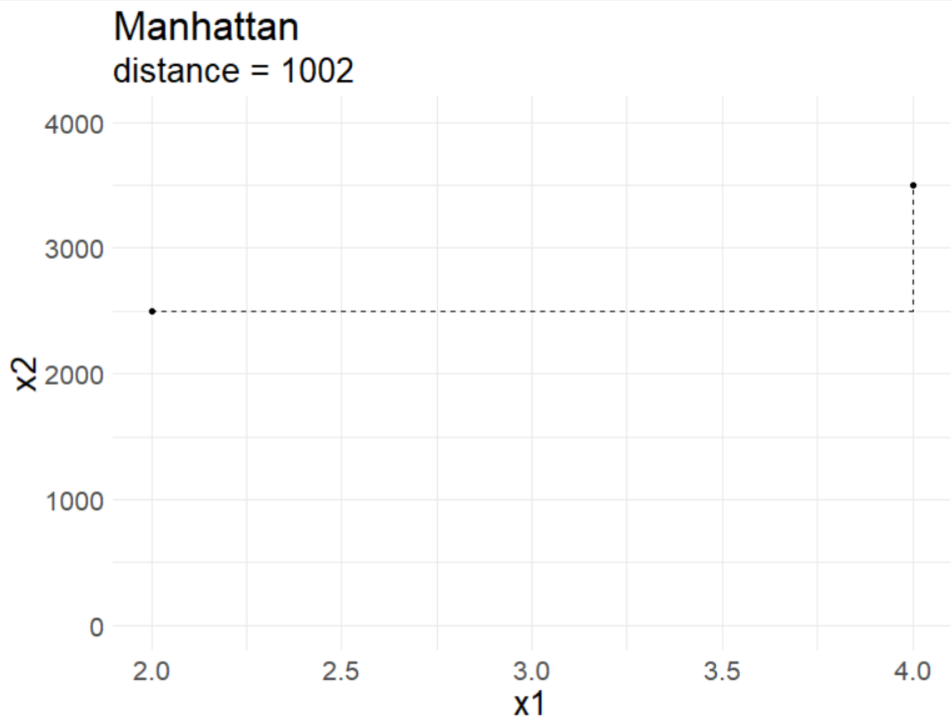
Euclidian

- as the crow flies
- common for continuous predictors

Manhattan

- city blocks
- common for binary predictors





dist_power

- Both Euclidian and Manhattan are special cases of Minkowski distance

<u>Minkowski</u>

$$\left(\sum_{j=1}^{P} \left|x_{aj} - x_{bj}\right|^{q}\right)^{\frac{1}{q}}$$

where q > 0 and $x_a$and x_a are individual predictors

when q = 2 we get Euclidian distance

<u>Euclidian</u>

$$\left(\sum_{j=1}^{P} \left(x_{aj} - x_{bj}\right)^{2}\right)^{\frac{1}{2}}$$

when q = 1 we get Manhattan distance

<u>Manhattan</u>

$$\left(\sum_{j=1}^{P} \left|x_{aj} - x_{bj}\right|\right)$$

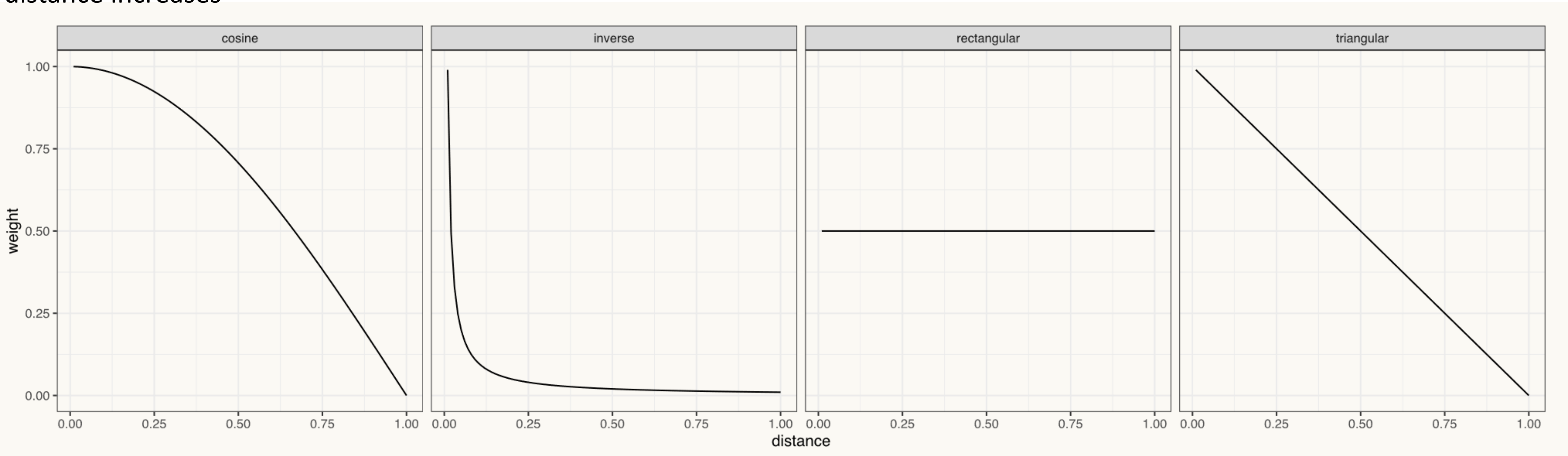
weight_func

- type of kernel function that weights the distances between samples
  1) "rectangular"
  2) "triangular"
  3) "epanechnikov"
  4) "biweight"
  5) "triweight"
  6) "cos"
  7) "inv"
  8) "gaussian"
  9) "rank"
  10) "optimal"

weight_func()

<u>cosine</u>
Slow decrease in weight as distance increases

<u>rectangular</u>
Uniform weight, regardless of distance



<u>inverse</u>
Sharp, immediate decrease in weight as distance increases, then relatively similar weight for those far away

<u>triangular</u>
Constant decrease in weight as distance increases

```r
set.seed(3000)
math <- read_csv(here::here("data", "train.csv")) %>%
  sample_frac(size = .02)

# 1 - Initial Split
set.seed(210)
math_split <- initial_split(math)

set.seed(210)
math_train <- training(math_split)
math_test  <- testing(math_split)


# 2 - Resample
set.seed(210)
math_cv <- vfold_cv(math_train)
```

“Recommended preprocessing

https://www.tmwr.org/pre-proc-table.html

model	dummy	zv	impute	decorrelate	normalize	transform
`nearest_neighbor()`	✓	✓	✓	◌	✓	✓

“Recommended preprocessing

https://www.tmwr.org/pre-proc-table.html

model	dummy	zv	impute	decorrelate	normalize	transform
`nearest_neighbor()`	✓	✓	✓	◌	✓	✓

step_dummy()

“Recommended preprocessing

https://www.tmwr.org/pre-proc-table.html

model	dummy	zv	impute	decorrelate	normalize	transform
`nearest_neighbor()`	✓	✓	✓	◌	✓	✓

step_zv()

“Recommended preprocessing

https://www.tmwr.org/pre-proc-table.html

model	dummy	zv	impute	decorrelate	normalize	transform
`nearest_neighbor()`	✓	✓	✓	◌	✓	✓

`step_bagimpute()`
`step_impute_linear()`
`step_knnimpute()`
`step_meanimpute()`
`step_medianimpute()`
`step_modeimpute()`
`step_lowerimpute()`
`step_rollimpute()`
`step_unknown()`

“Recommended preprocessing

https://www.tmwr.org/pre-proc-table.html

model	dummy	zv	impute	decorrelate	normalize	transform
`nearest_neighbor()`	✓	✓	✓	◌	✓	✓

`step_corr()`

“Recommended preprocessing

https://www.tmwr.org/pre-proc-table.html

model	dummy	zv	impute	decorrelate	normalize	transform
`nearest_neighbor()`	✓	✓	✓	◌	✓	✓

```
step_normalize()
step_center()
step_scale()
```

“Recommended preprocessing

https://www.tmwr.org/pre-proc-table.html

model	dummy	zv	impute	decorrelate	normalize	transform
`nearest_neighbor()`	✓	✓	✓	◌	✓	✓

`step_BoxCox()`
`step_YeoJohnson()`
`step_log()`
`step_sqrt()`
`step_inverse()`

more…

- We’re going to fit a classification model, so let’s take a look at our outcome.

```
math_train %>%
  tabyl(classification)
 classification   n   percent
              1 978 0.3441239
              2 732 0.2575651
              3 637 0.2241379
              4 495 0.1741731

#  Preprocess

knn1_rec <-
  recipe(
    classification ~ enrl_grd + lat + lon,
    data = math_train
  ) %>%
  step_mutate(classification = ifelse(classification < 3, "below", "proficient")) %>%
  step_mutate(enrl_grd = factor(enrl_grd)) %>%
  step_meanimpute(lat, lon) %>%
  step_unknown(enrl_grd) %>%
  step_dummy(enrl_grd) %>%
  step_normalize(lat, lon)
```

- We’re going to fit a classification model, so let’s take a look at our outcome.

```
math_train %>%
  tabyl(classification)
 classification   n   percent
              1 973 0.3423645
              2 730 0.2568614
              3 631 0.2220267
              4 508 0.1787474
```

```
#  Preprocess

knn1_rec <-
  recipe(
    classification ~ enrl_grd + lat + lon,
    data = math_train
  ) %>%
  step_mutate(classification = ifelse(classification < 3, "below", "proficient")) %>%
  step_mutate(enrl_grd = factor(enrl_grd)) %>%
  step_meanimpute(lat, lon) %>%
  step_unknown(enrl_grd) %>%
  step_dummy(enrl_grd) %>%
  step_normalize(lat, lon)
```

- We're going to fit a classification model, so let's take a look at our outcome.

```
math_train %>%
  tabyl(classification)
 classification   n   percent
              1 973 0.3423645
              2 730 0.2568614
              3 631 0.2220267
              4 508 0.1787474
```

```
#  Preprocess

knn1_rec <-
  recipe(
    classification ~ enrl_grd + lat + lon,
    data = math_train
  ) %>%
  step_mutate(classification = ifelse(classification < 3, "below", "proficient")) %>%
  step_mutate(enrl_grd = factor(enrl_grd)) %>%
  step_meanimpute(lat, lon) %>%
  step_unknown(enrl_grd) %>%
  step_dummy(enrl_grd) %>%
  step_normalize(lat, lon)
```

- We're going to fit a classification model, so let's take a look at our outcome.

```
math_train %>%
  tabyl(classification)
 classification   n   percent
              1 973 0.3423645
              2 730 0.2568614
              3 631 0.2220267
              4 508 0.1787474

#  Preprocess

knn1_rec <-
  recipe(
    classification ~ enrl_grd + lat + lon,
    data = math_train
  ) %>%
  step_mutate(classification = ifelse(classification < 3, "below", "proficient")) %>%
  step_mutate(enrl_grd = factor(enrl_grd)) %>%
  step_meanimpute(lat, lon) %>%
  step_unknown(enrl_grd) %>%
  step_dummy(enrl_grd) %>%
  step_normalize(lat, lon)
```

- We’re going to fit a classification model, so let’s take a look at our outcome.

```
math_train %>%
  tabyl(classification)
 classification   n   percent
              1 973 0.3423645
              2 730 0.2568614
              3 631 0.2220267
              4 508 0.1787474
```

```
#  Preprocess

knn1_rec <-
  recipe(
    classification ~ enrl_grd + lat + lon,
    data = math_train
  ) %>%
  step_mutate(classification = ifelse(classification < 3, "below", "proficient")) %>%
  step_mutate(enrl_grd = factor(enrl_grd)) %>%
  step_meanimpute(lat, lon) %>%
  step_unknown(enrl_grd) %>%
  step_dummy(enrl_grd) %>%
  step_normalize(lat, lon)
```

- We're going to fit a classification model, so let's take a look at our outcome.

```
math_train %>%
  tabyl(classification)
 classification   n   percent
              1 973 0.3423645
              2 730 0.2568614
              3 631 0.2220267
              4 508 0.1787474

#  Preprocess

knn1_rec <-
  recipe(
    classification ~ enrl_grd + lat + lon,
    data = math_train
  ) %>%
  step_mutate(classification = ifelse(classification < 3, "below", "proficient")) %>%
  step_mutate(enrl_grd = factor(enrl_grd)) %>%
  step_meanimpute(lat, lon) %>%
  step_unknown(enrl_grd) %>%
  step_dummy(enrl_grd) %>%
  step_normalize(lat, lon)
```

- We’re going to fit a classification model, so let’s take a look at our outcome.

```
math_train %>%
  tabyl(classification)
 classification   n   percent
              1 973 0.3423645
              2 730 0.2568614
              3 631 0.2220267
              4 508 0.1787474

#  Preprocess

knn1_rec <-
  recipe(
    classification ~ enrl_grd + lat + lon,
    data = math_train
  ) %>%
  step_mutate(classification = ifelse(classification < 3, "below", "proficient")) %>%
  step_mutate(enrl_grd = factor(enrl_grd)) %>%
  step_meanimpute(lat, lon) %>%
  step_unknown(enrl_grd) %>%
  step_dummy(enrl_grd) %>%
  step_normalize(lat, lon)
```

- We’re going to fit a classification model, so let’s take a look at our outcome.

```
math_train %>%
  tabyl(classification)
 classification   n   percent
              1 973 0.3423645
              2 730 0.2568614
              3 631 0.2220267
              4 508 0.1787474

#  Preprocess

knn1_rec <-
  recipe(
    classification ~ enrl_grd + lat + lon,
    data = math_train
  ) %>%
  step_mutate(classification = ifelse(classification < 3, "below", "proficient")) %>%
  step_mutate(enrl_grd = factor(enrl_grd)) %>%
  step_meanimpute(lat, lon) %>%
  step_unknown(enrl_grd) %>%
  step_dummy(enrl_grd) %>%
  step_normalize(lat, lon)
```

```r
# 3 - Set Model
## KNN

knn1_mod <- nearest_neighbor() %>%
  set_engine("kknn") %>%
  set_mode(“classification")

translate(knn1_mod)
```

`translate()` will translate a model specification into a code object that is specific to a particular engine

```
# 3 - Set Model
## KNN

knn1_mod <- nearest_neighbor() %>%
  set_engine("kknn") %>%
  set_mode("classification")

translate(knn1_mod)

K-Nearest Neighbor Model Specification (classification)

Computational engine: kknn

Model fit template:
kknn::train.kknn(formula = missing_arg(), data = missing_arg(),
    ks = min_rows(5, data, 5))
```

`min_rows()`

- For some tuning parameters, the range of values depend on the data dimensions. This function checks the possible range of the data and adjust them if needed (with a warning).

`min_rows(num_rows, source, offset)`

- `num_rows:` set by you
- `data:` a data frame for the data to be used in the fit
- `offset:` number subtracted off of the number of rows available in the data

```r
# 4 - Tune
## Let's run the default tuned KNN model for `neighbors`, `weight_func`, and `dist_power`

knn1_mod <- knn1_mod %>%
  set_args(neighbors = tune(),
           weight_func = tune(),
           dist_power = tune())

translate(knn1_mod)
```

```
K-Nearest Neighbor Model Specification (classification)

Main Arguments:
  neighbors = tune()
  weight_func = tune()
  dist_power = tune()

Computational engine: kknn

Model fit template:
kknn::train.kknn(formula = missing_arg(), data = missing_arg(),
    ks = min_rows(tune(), data, 5), kernel = tune(), distance = tune())
```

Parallel Processing (quickly)

- `{parallel}`
  - used for parallel processing
  - `detectCores()` will tell you how many cores you have access to
  - `makeCluster()` creates a set of copies of R running in parallel
- `{doParallel}`
  - provides a parallel backend using the `{parallel}` package
  - `registerDoParallel()` is used to register the parallel backend with the `{foreach}` package
    - `{foreach}` supports parallel execution; it can execute repeated operations on multiple processors/cores on your computer, or on multiple nodes of a cluster

```r
parallel::detectCores()

tic()
cl <- parallel::makeCluster(8)

doParallel::registerDoParallel(cl)

knn1_res <- tune::tune_grid(
  knn1_mod,
  preprocessor = knn1_rec,
  resamples = math_cv,
  control = tune::control_resamples(save_pred = TRUE)
)

parallel::stopCluster(cl)
toc()
```

```r
parallel::detectCores()

tic()
cl <- parallel::makeCluster(8)

doParallel::registerDoParallel(cl)

knn1_res <- tune::tune_grid(
  knn1_mod,
  preprocessor = knn1_rec,
  resamples = math_cv,
  control = tune::control_resamples(save_pred = TRUE)
)

parallel::stopCluster(cl)
toc()
```

```r
parallel::detectCores()

tic()
cl <- parallel::makeCluster(8)

doParallel::registerDoParallel(cl)

knn1_res <- tune::tune_grid(
  knn1_mod,
  preprocessor = knn1_rec,
  resamples = math_cv,
  control = tune::control_resamples(save_pred = TRUE)
)

parallel::stopCluster(cl)
toc()
```

```r
parallel::detectCores()

tic()
cl <- parallel::makeCluster(8)

doParallel::registerDoParallel(cl)

knn1_res <- tune::tune_grid(
  knn1_mod,
  preprocessor = knn1_rec,
  resamples = math_cv,
  control = tune::control_resamples(save_pred = TRUE)
)

parallel::stopCluster(cl)
toc()

# without clustering: 363.86 sec elapsed
# with clustering: 63.78 sec elapsed
```

```r
parallel::detectCores()
tic()
cl <- parallel::makeCluster(8)

doParallel::registerDoParallel(cl)

knn1_res <- tune::tune_grid(
  knn1_mod,
  preprocessor = knn1_rec,
  resamples = math_cv,
  control = tune::control_resamples(save_pred = TRUE)
)

parallel::stopCluster(cl)
toc()
```

```
knn1_res %>%
  select(.predictions) %>%
  unnest()
```

```
# A tibble: 28,420 x 9
   .pred_below .pred_proficient  .row neighbors weight_func dist_power .pred_class classification .config
         <dbl>            <dbl> <int>     <int> <chr>            <dbl> <fct>       <fct>          <chr>
 1       0.634            0.366     4        10 biweight         0.805 below       proficient     Model01
 2       0.259            0.741     5        10 biweight         0.805 proficient  below          Model01
 3       0.677            0.323     7        10 biweight         0.805 below       below          Model01
 4       0.726            0.274    47        10 biweight         0.805 below       below          Model01
 5       0.740            0.260    91        10 biweight         0.805 below       below          Model01
 6       0.663            0.337   105        10 biweight         0.805 below       below          Model01
 7       0.527            0.473   122        10 biweight         0.805 below       below          Model01
 8       0.739            0.261   132        10 biweight         0.805 below       below          Model01
 9       0.280            0.720   136        10 biweight         0.805 proficient  below          Model01
10       0.519            0.481   138        10 biweight         0.805 below       below          Model01
# ... with 28,410 more rows
```

- The first two columns represent class probabilities for our two outcome classes
- The `.pred_class` column represents the class predicted by the model (class with highest probability)
  - Thus, most classification models can generate "hard" and "soft" predictions for models
  - The class predictions are usually created by thresholding some numeric output of the model (e.g. a class probability) or by choosing the largest value
- The `classification` column is the observed outcome class (truth)

```r
knn1_res %>%
  collect_predictions()
```

```
# A tibble: 28,420 x 10
   id     .pred_below .pred_proficient  .row neighbors weight_func dist_power .pred_class classification .config
   <chr>        <dbl>            <dbl> <int>     <int> <chr>            <dbl> <fct>       <fct>          <chr>
 1 Fold01       0.634            0.366     4        10 biweight         0.805 below       proficient     Model01
 2 Fold01       0.259            0.741     5        10 biweight         0.805 proficient  below          Model01
 3 Fold01       0.677            0.323     7        10 biweight         0.805 below       below          Model01
 4 Fold01       0.726            0.274    47        10 biweight         0.805 below       below          Model01
 5 Fold01       0.740            0.260    91        10 biweight         0.805 below       below          Model01
 6 Fold01       0.663            0.337   105        10 biweight         0.805 below       below          Model01
 7 Fold01       0.527            0.473   122        10 biweight         0.805 below       below          Model01
 8 Fold01       0.739            0.261   132        10 biweight         0.805 below       below          Model01
 9 Fold01       0.280            0.720   136        10 biweight         0.805 proficient  below          Model01
10 Fold01       0.519            0.481   138        10 biweight         0.805 below       below          Model01
# ... with 28,410 more rows
```

```
knn1_res %>%
  collect_metrics(summarize = FALSE)
```

```
# A tibble: 200 x 8
   id     neighbors weight_func  dist_power .metric  .estimator .estimate .config
   <chr>      <int> <chr>             <dbl> <chr>    <chr>          <dbl> <chr>
 1 Fold01        10 biweight          0.805 accuracy binary         0.572 Model01
 2 Fold01        10 biweight          0.805 roc_auc  binary         0.593 Model01
 3 Fold01         2 cos               1.84  accuracy binary         0.565 Model02
 4 Fold01         2 cos               1.84  roc_auc  binary         0.534 Model02
 5 Fold01        12 epanechnikov      0.222 accuracy binary         0.565 Model03
 6 Fold01        12 epanechnikov      0.222 roc_auc  binary         0.568 Model03
 7 Fold01        14 gaussian          0.316 accuracy binary         0.568 Model04
 8 Fold01        14 gaussian          0.316 roc_auc  binary         0.581 Model04
 9 Fold01         5 inv               0.986 accuracy binary         0.572 Model05
10 Fold01         5 inv               0.986 roc_auc  binary         0.559 Model05
```

```r
knn1_res %>%
  collect_metrics(summarize = FALSE) %>%
  distinct(neighbors, weight_func, dist_power)
```

```
# A tibble: 10 x 3
   neighbors weight_func  dist_power
       <int> <chr>             <dbl>
 1        10 biweight          0.805
 2         2 cos               1.84
 3        12 epanechnikov      0.222
 4        14 gaussian          0.316
 5         5 inv               0.986
 6         7 optimal           1.38
 7        13 rank              1.23
 8         3 rectangular       1.59
 9         6 triangular        1.74
10         8 triweight         0.569
```

- There are 10 unique values because in `tune_grid()`, the default argument is `grid = 10`

Performance estimates

```
knn1_res %>%
  show_best(metric = "roc_auc", n = 10)
```

```
# A tibble: 10 x 9
   neighbors weight_func  dist_power .metric .estimator  mean     n std_err .config
       <int> <chr>             <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
 1        13 rank              1.23  roc_auc binary     0.585    10 0.0154  Model07
 2        14 gaussian          0.316 roc_auc binary     0.580    10 0.0147  Model04
 3         7 optimal           1.38  roc_auc binary     0.580    10 0.0111  Model06
 4        10 biweight          0.805 roc_auc binary     0.579    10 0.0124  Model01
 5         5 inv               0.986 roc_auc binary     0.578    10 0.0118  Model05
 6         6 triangular        1.74  roc_auc binary     0.574    10 0.0119  Model09
 7        12 epanechnikov      0.222 roc_auc binary     0.574    10 0.0157  Model03
 8         8 triweight         0.569 roc_auc binary     0.570    10 0.00956 Model10
 9         3 rectangular       1.59  roc_auc binary     0.565    10 0.0118  Model08
10         2 cos               1.84  roc_auc binary     0.559    10 0.0132  Model02
```

Performance estimates "by hand”

```
knn1_res$.metrics %>%
  bind_rows(.id = "fold") %>%
  filter(`.metric` == "roc_auc") %>%
  group_by(neighbors, weight_func, dist_power) %>%
  summarize(mean = mean(`.estimate`),
            se = sd(`.estimate`)/sqrt(n())) %>%
  arrange(desc(mean))
```

```
# A tibble: 10 x 5
# Groups:   neighbors, weight_func [10]
   neighbors weight_func  dist_power  mean      se
       <int> <chr>             <dbl> <dbl>   <dbl>
 1        13 rank              1.23  0.585 0.0154
 2        14 gaussian          0.316 0.580 0.0147
 3         7 optimal           1.38  0.580 0.0111
 4        10 biweight          0.805 0.579 0.0124
 5         5 inv               0.986 0.578 0.0118
 6         6 triangular        1.74  0.574 0.0119
 7        12 epanechnikov      0.222 0.574 0.0157
 8         8 triweight         0.569 0.570 0.00956
 9         3 rectangular       1.59  0.565 0.0118
10         2 cos               1.84  0.559 0.0132
```

show_best() & select_best()

```
knn1_res %>%
  show_best(metric = "roc_auc", n = 1)
```

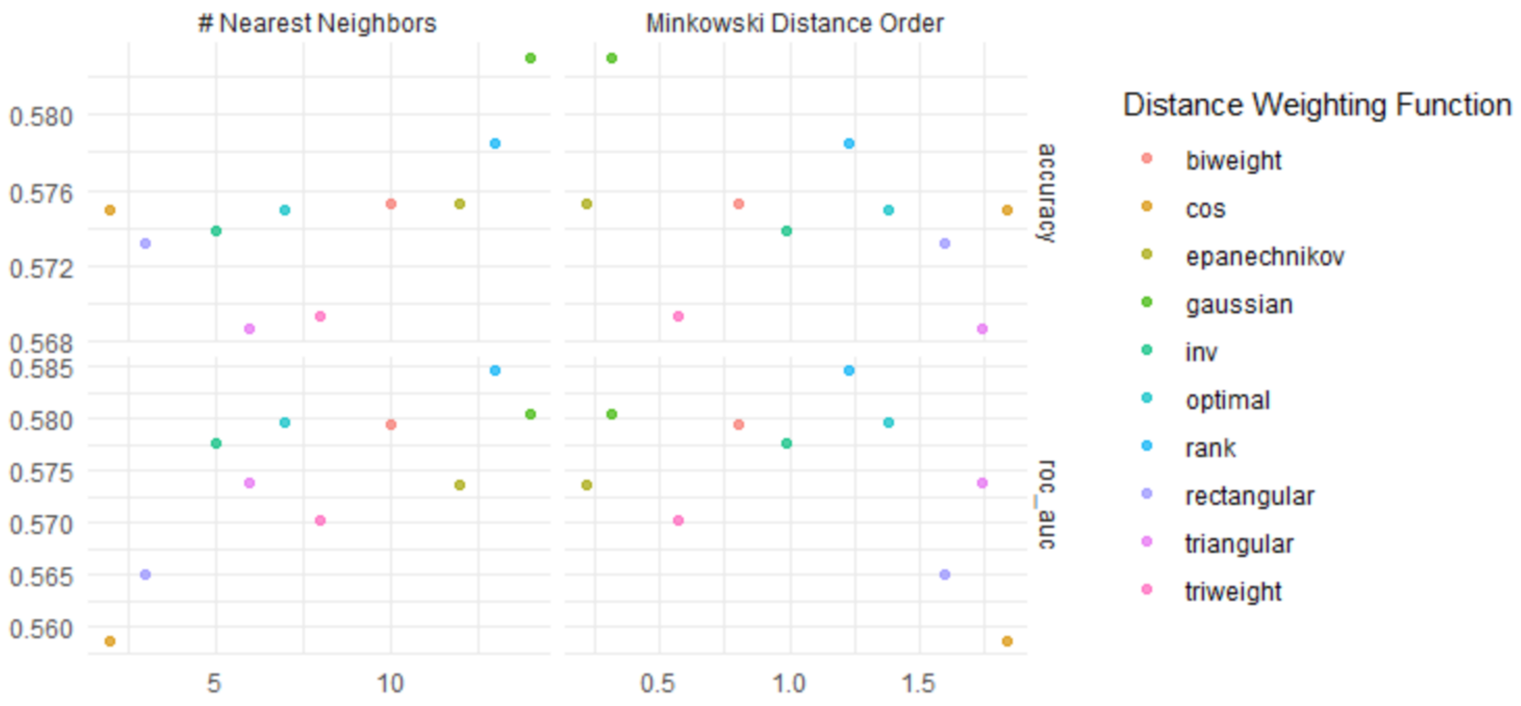
```
 A tibble: 1 x 9
  neighbors weight_func dist_power .metric .estimator  mean     n std_err .config
      <int> <chr>            <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
1        13 rank              1.23 roc_auc binary     0.585    10  0.0154 Model07
```

```
knn1_res %>%
  select_best(metric = "roc_auc")
```

```
# A tibble: 1 x 4
  neighbors weight_func dist_power .config
      <int> <chr>            <dbl> <chr>
1        13 rank              1.23 Model07
```

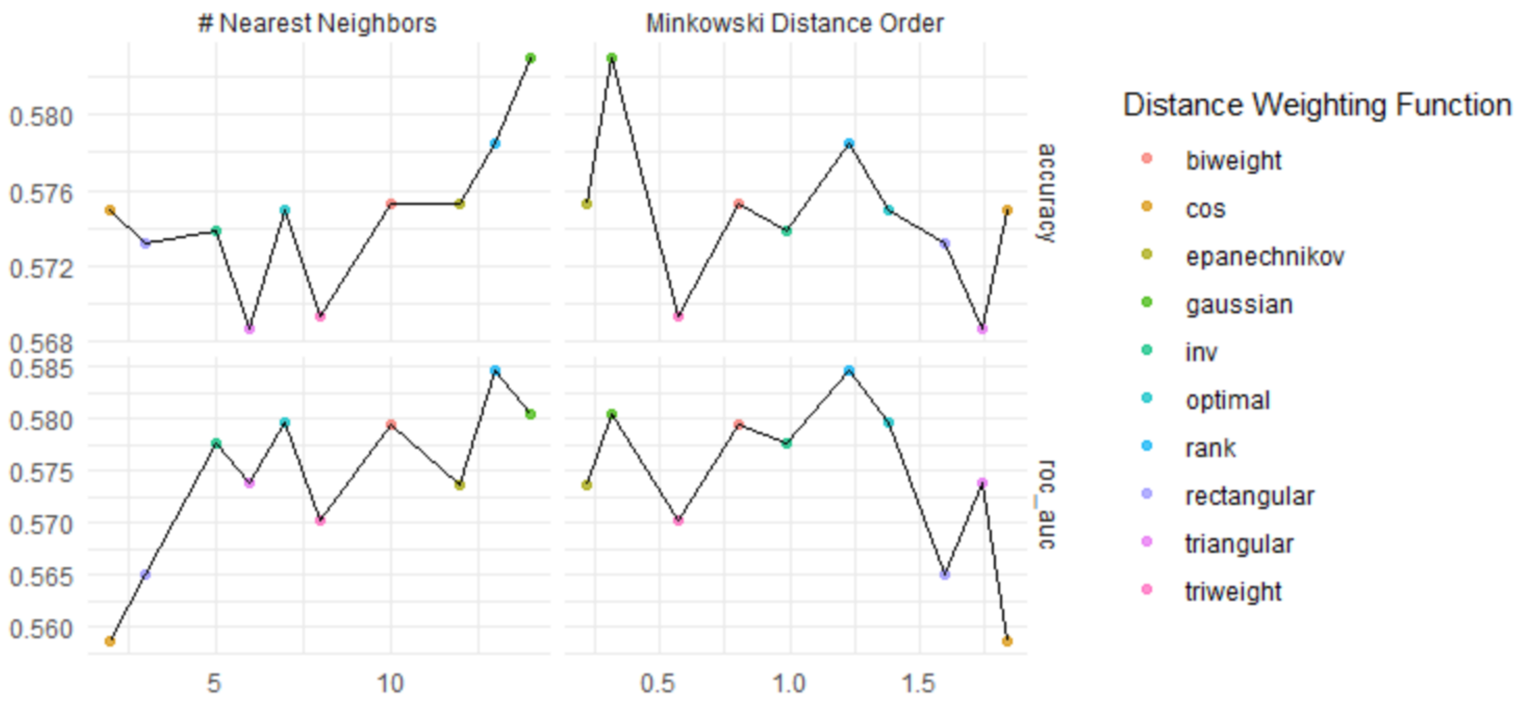
autoplot()

```
knn1_res %>%
  autoplot()
```



autoplot()

```
knn1_res %>%
  autoplot() +
  geom_line()
```



autoplot()

```
autoplot(
  object,
  type = c("marginals",
           "parameters",
           "performance"),
  metric = NULL,
  width = NULL,
  ...
)
```

autoplot()

```
autoplot(
  object,
  type = c("marginals",
           "parameters",
           "performance"),
  metric = NULL,
  width = NULL,
  ...
)
```

a tibble or results from `tune_grid()` or `tune_bayes()`

autoplot()

```
autoplot(
  object,
  type = c("marginals",
           "parameters",
           "performance"),
  metric = NULL,
  width = NULL,
  ...
)
```

`tune_grid()`

"marginals" = for a plot of each predictor versus performance

"parameters“ = each parameter versus search iteration

`tune_bayes()`

"performance" = performance versus iteration

autoplot()

```
autoplot(
  object,
  type = c("marginals",
           "parameters",
           "performance"),
  metric = NULL,
  width = NULL,
  ...
)
```

which `metric` to plot
(default `NULL` is all metrics shown via facets)

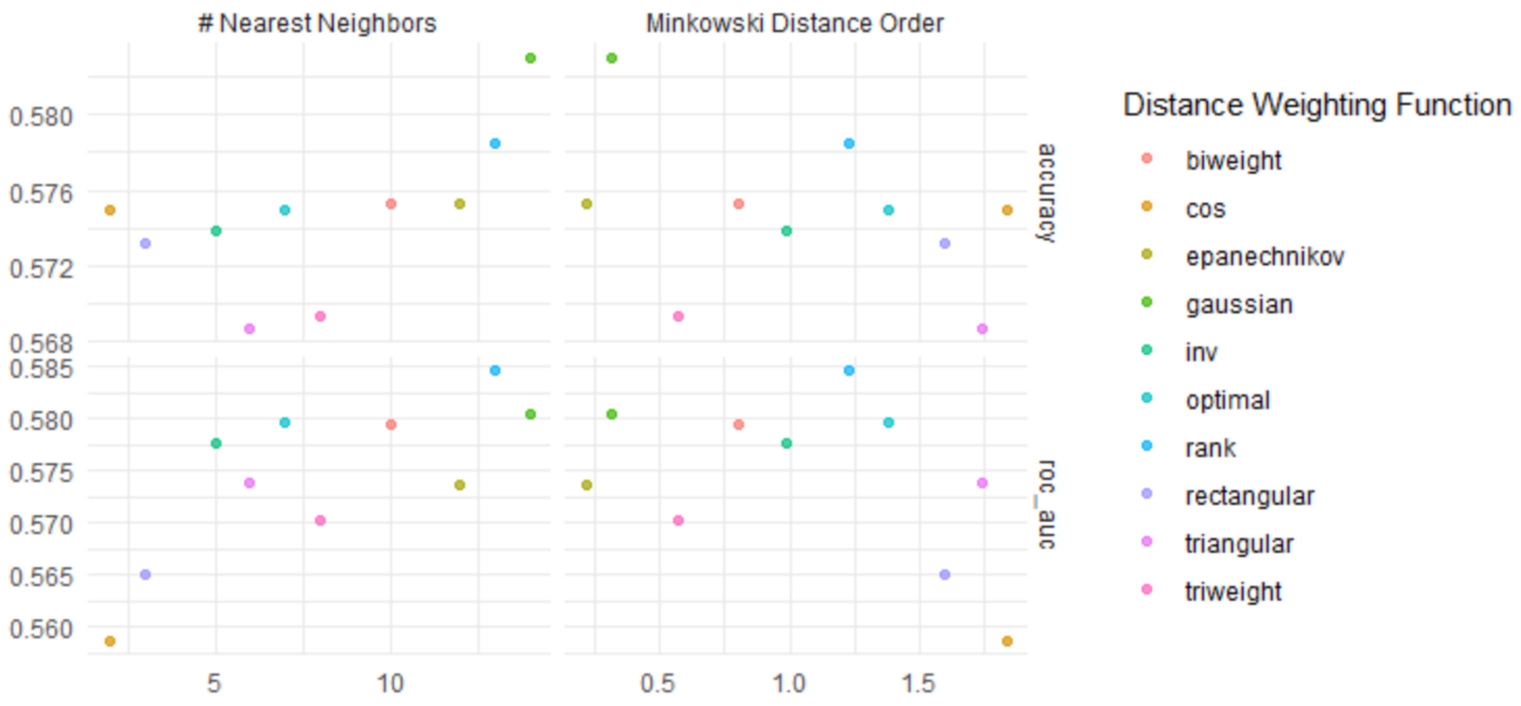
autoplot()

```
autoplot(
  object,
  type = c("marginals",
           "parameters",
           "performance"),
  metric = NULL,
  width = NULL,
  ...
)
```

For `type = "perfomance“`
A number for the width of the confidence interval bars (where zero prevents them from being shown)

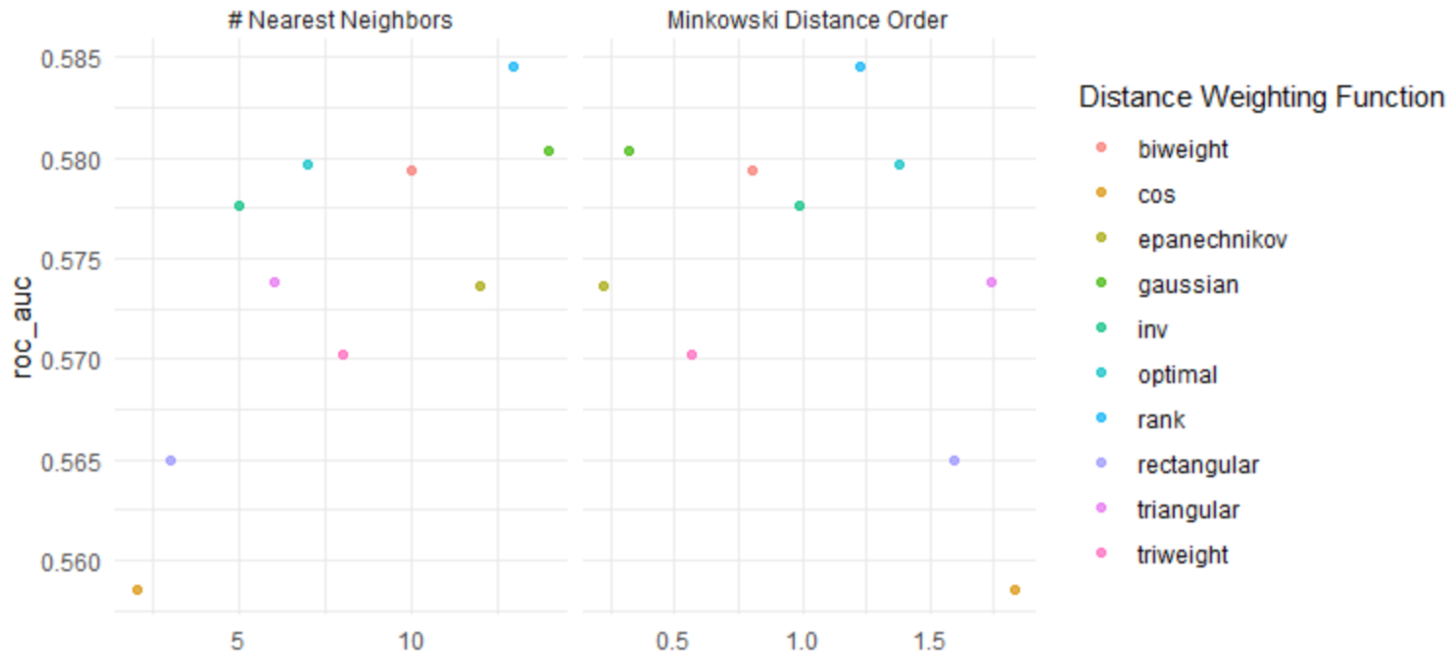
autoplot()

```
knn1_res %>%
  autoplot()
```



autoplot()

```
knn1_res %>%
  autoplot(metric = "roc_auc")
```

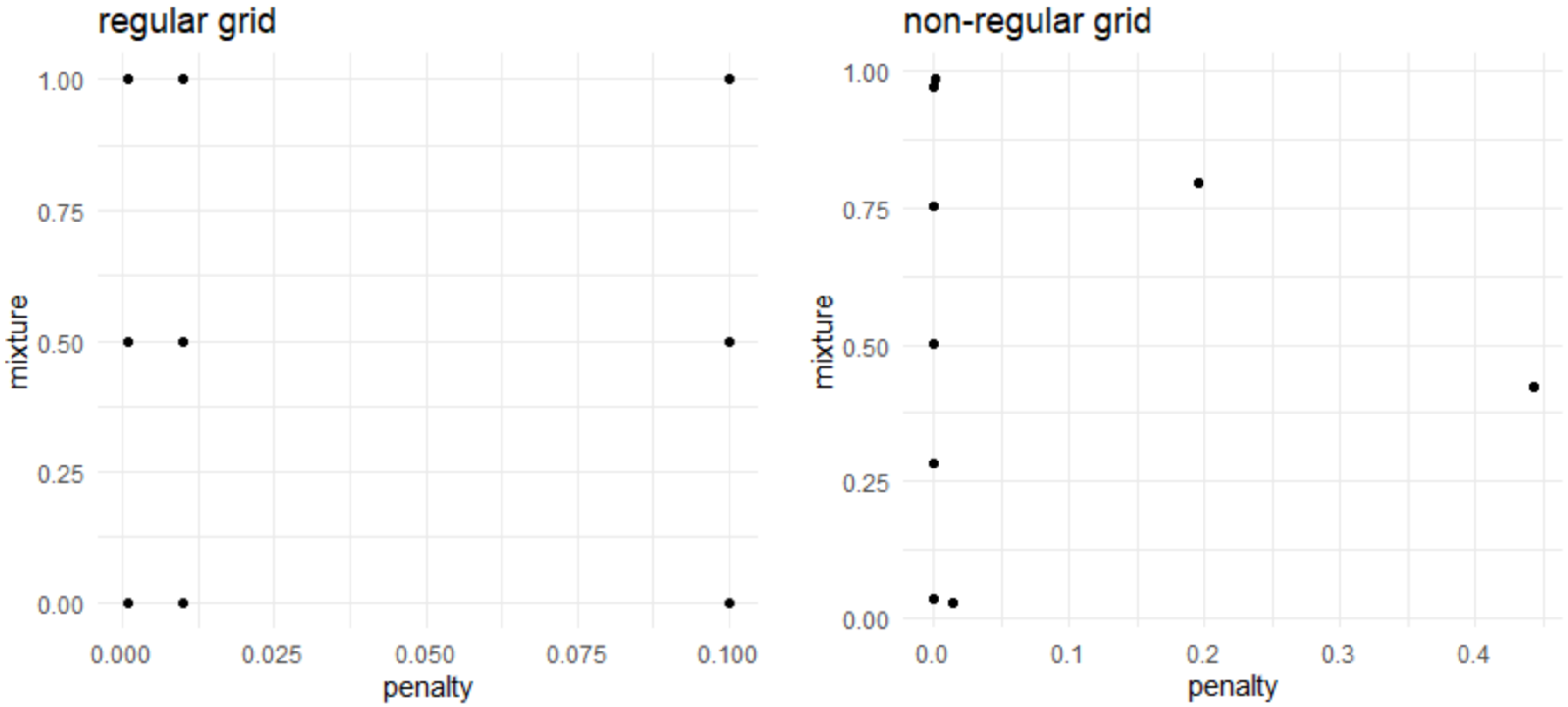


More grids

non-regular grids

Regular vs. Non-regular grids

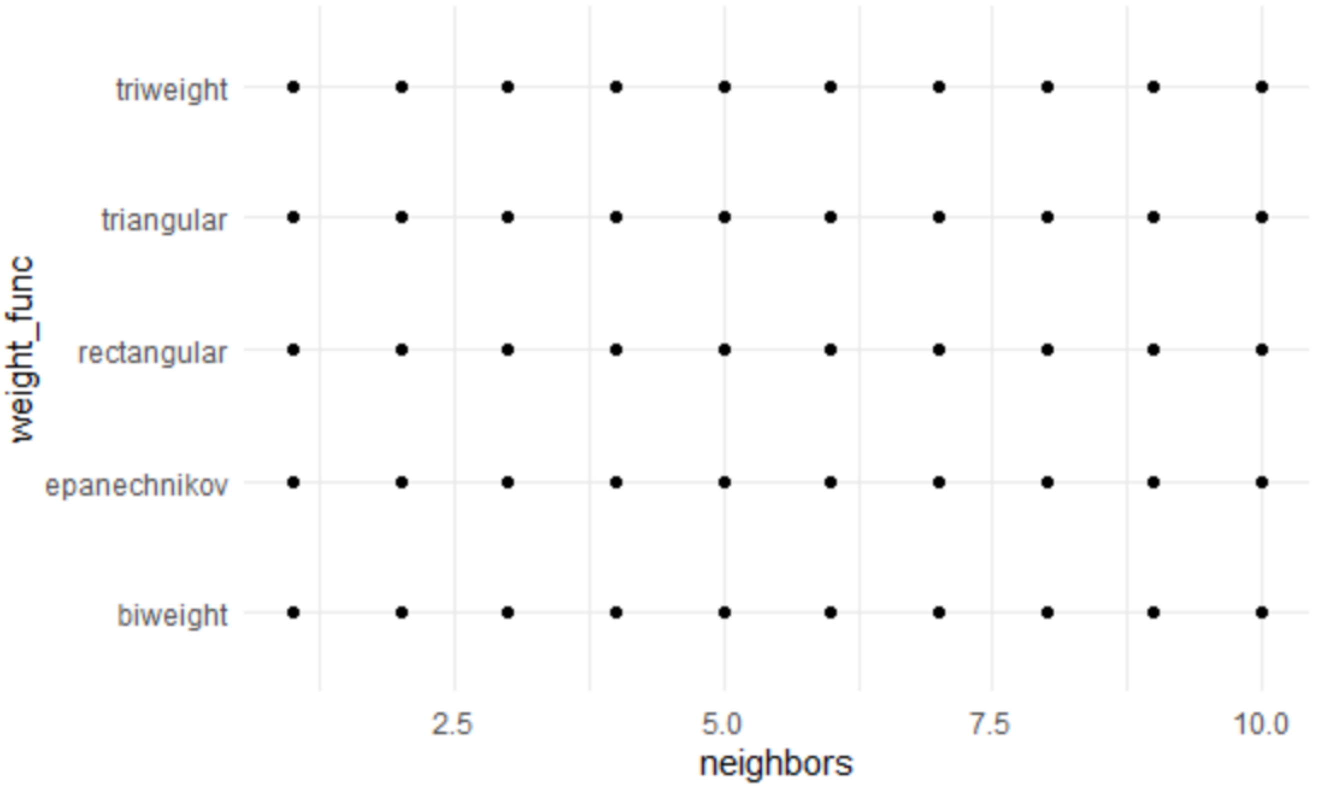
- Regular grid
  - a known, pre-defined set of tuning parameter values
  - the number of values don't have to be the same per parameter
  - Quantitative and qualitative parameters can be combined
  - As the number of parameters increases, so does the burden of dimensionality
  - Thought to be inefficient but not in all cases
- Non-regular grids (or random grids)
  - define a range of possible values for each parameter and randomly sample the multidimensional space enough times to cover a reasonable amount
  - beneficial when there are a large number of tuning parameters and there is no *a priori* notion of which values should be used
  - A large grid may be inefficient to search, especially if the profile has a fairly stable pattern with little change over some range of the parameter
  - Good for neural networks and gradient boosting machines


regular grid
mixture
penalty
1.00
0.75
0.50
0.25
0.00
0.000
0.025
0.050
0.075
0.100
non-regular grid
mixture
penalty
1.00
0.75
0.50
0.25
0.00
0.0
0.1
0.2
0.3
0.4

Regular grids

Let’s look at a regular grid

```
knn_params <- parameters(neighbors(), weight_func())
knn_reg_grid <- grid_regular(neighbors(), weight_func(), levels = c(15, 5))
dim(knn_reg_grid)
[1] 50  2
```



A closer look at `knn_params`

```
str(knn_params)
```

```
tibble [2 x 6] (S3: parameters/tbl_df/tbl/data.frame)
 $ name        : chr [1:2] "neighbors" "weight_func"
 $ id          : chr [1:2] "neighbors" "weight_func"
 $ source      : chr [1:2] "list" "list"
 $ component   : chr [1:2] "unknown" "unknown"
 $ component_id: chr [1:2] "unknown" "unknown"
 $ object      :List of 2
  ..$ :List of 7
  .. ..$ type     : chr "integer"
  .. ..$ range    :List of 2
  .. .. ..$ lower: int 1
  .. .. ..$ upper: int 10
```

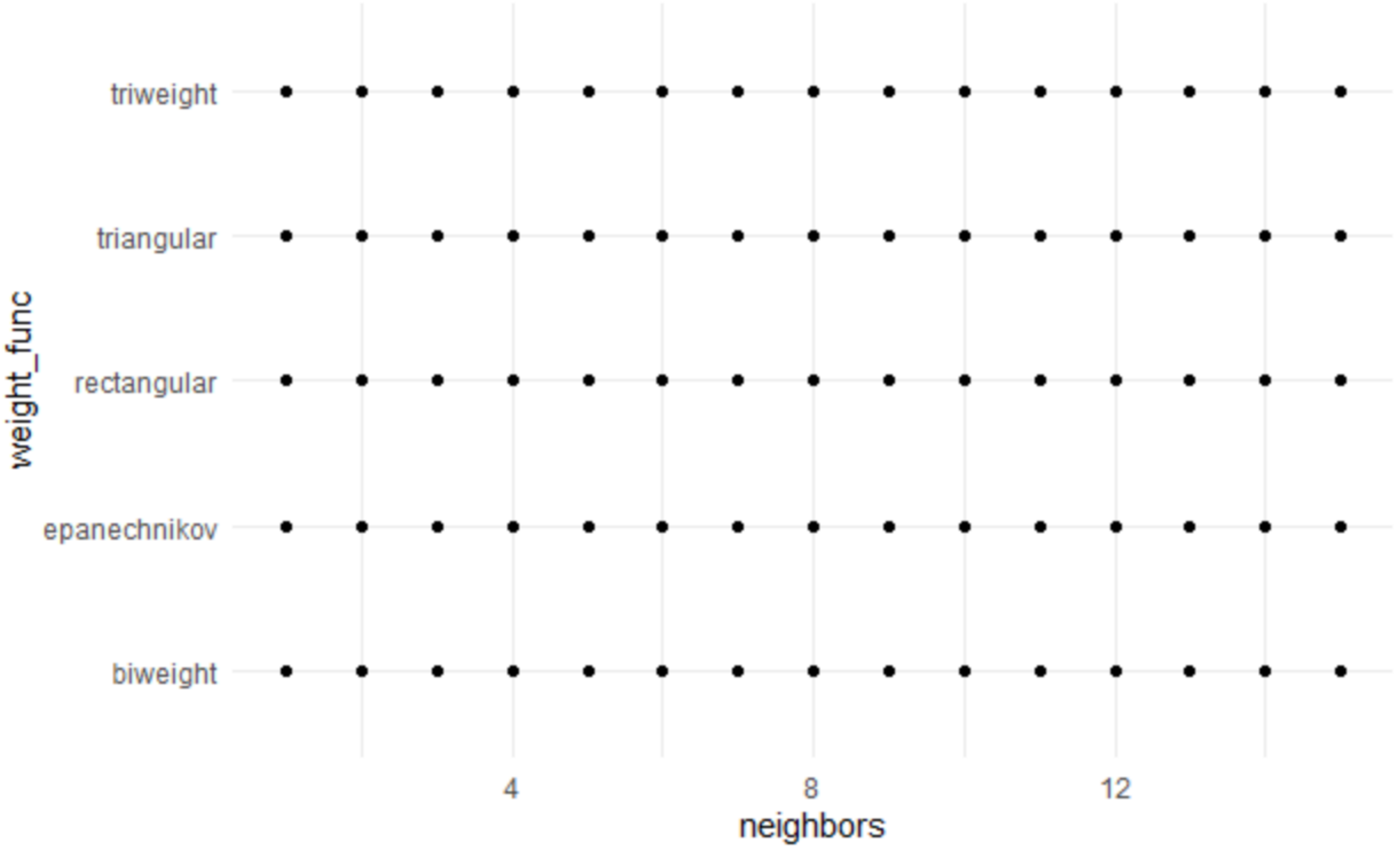
Two ways to address this

(1) use the arguments within the hyperparameters

```
?neighbors()
neighbors(range = c(1L, 10L), trans = NULL)

?weight_func()
weight_func(values = values_weight_func)

values_weight_func
"rectangular"  "triangular"   "epanechnikov" "biweight"
"triweight"  "cos"  "inv"  "gaussian"  "rank"  "optimal"

knn_params <- parameters(neighbors(range = c(1, 15)),
                         weight_func(values = values_weight_func[1:5]))

knn_reg_grid <- grid_regular(knn_params, levels = c(15, 5))
```

(1) use the arguments within the hyperparameters

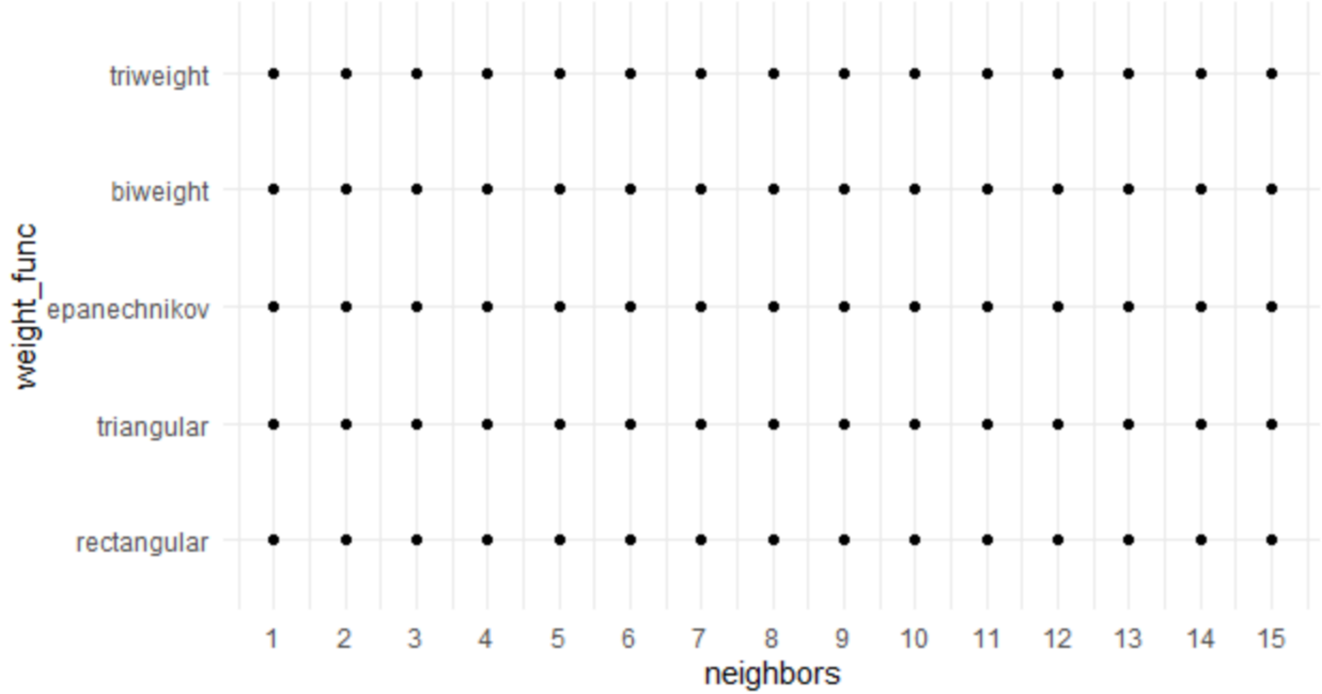
```
knn_reg_grid %>%
  ggplot(aes(neighbors, weight_func)) +
  geom_point()
```



(2) Let's make our own

- Complete flexibility

```
knn_reg_grid_man <- expand.grid(
            neighbors = c(1:15),
            weight_func = values_weight_func[1:5])
```



Non-regular grids

Non-regular grids

- There are two main methods to make non-regular grids
  - **Random grids** uniformly sample the parameter space (that might already be on a different scale)
  - **Space-filling designs (SFD)** are based on statistical experimental design principles and try to keep candidate values away from one another while encompassing the entire parameter space
- There's no real downside to using SFD, so we will focus mostly on these

grid_max_entropy()

```
grid_max_entropy(
  x,
  ...,
  size = 3,
  original = TRUE,
  variogram_range = 0.5,
  iter = 1000
)
```

`x` : A `param` object, list, or `parameters`

`...` : One or more param objects (e.g., `penalty()`). Cannot have `unknown()` values in the parameter ranges or values

`size`: A single integer for the total number of parameter value combinations returned

`original`: A logical: should the parameters be in the original units or in the transformed space (if any)?

`variogram_range` : A numeric value greater than zero. Larger values reduce the likelihood of empty regions in the parameter space.

`iter` : An integer for the maximum number of iterations used to find a good design.

grid_max_entropy()

```
grid_max_entropy(
  x,
  ...,
  size = 3,
  original = TRUE,
  variogram_range = 0.5,
  iter = 1000
)
```

`x` : A `param` object, list, or `parameters`

`...` : One or more param objects (e.g., `penalty()`). Cannot have `unknown()` values in the parameter ranges or values

`size`: A single integer for the total number of parameter value combinations returned

`original`: A logical: should the parameters be in the original units or in the transformed space (if any)?

`variogram_range` : A numeric value greater than zero. Larger values reduce the likelihood of empty regions in the parameter space.

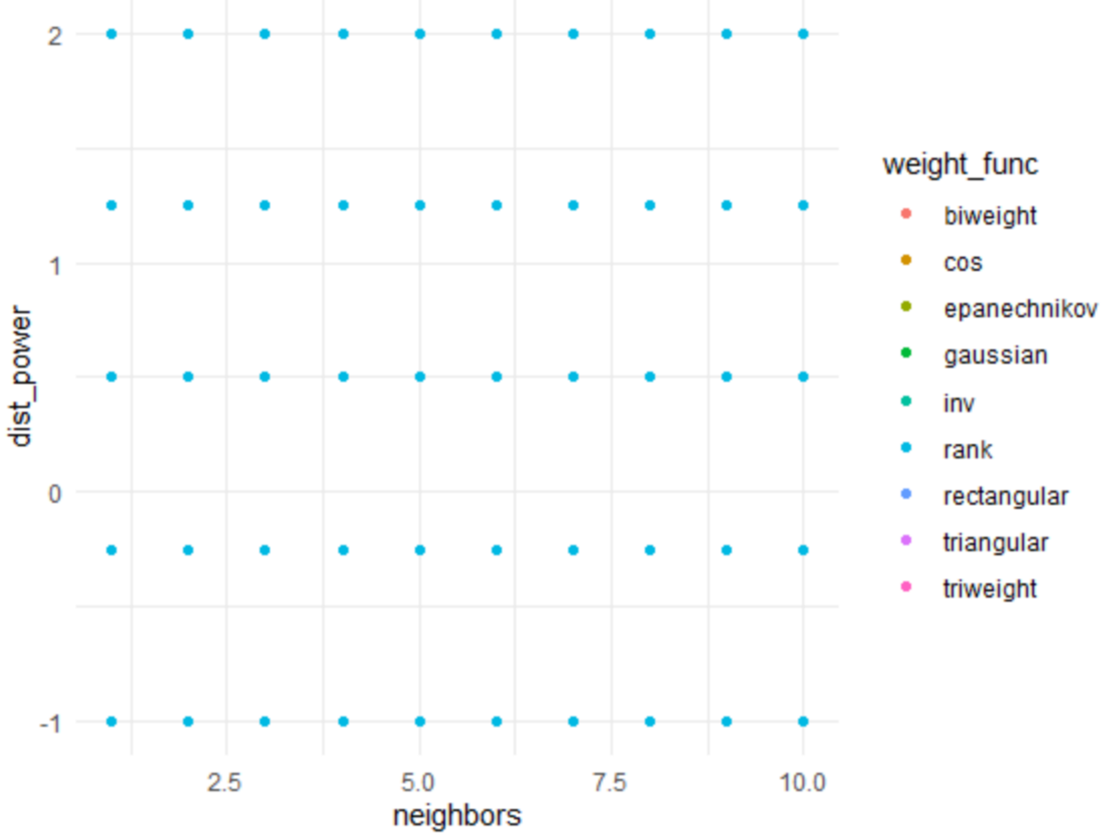
`iter` : An integer for the maximum number of iterations used to find a good design.

grid_regular()

knn_params <- parameters(neighbors(), weight_func(), dist_power())

knn_grid_reg <- grid_regular(knn_params, levels = c(10, 9, 5))

```
knn_grid_reg %>%
  ggplot(aes(neighbors, dist_power)) +
  geom_point(aes(color = weight_func))
```

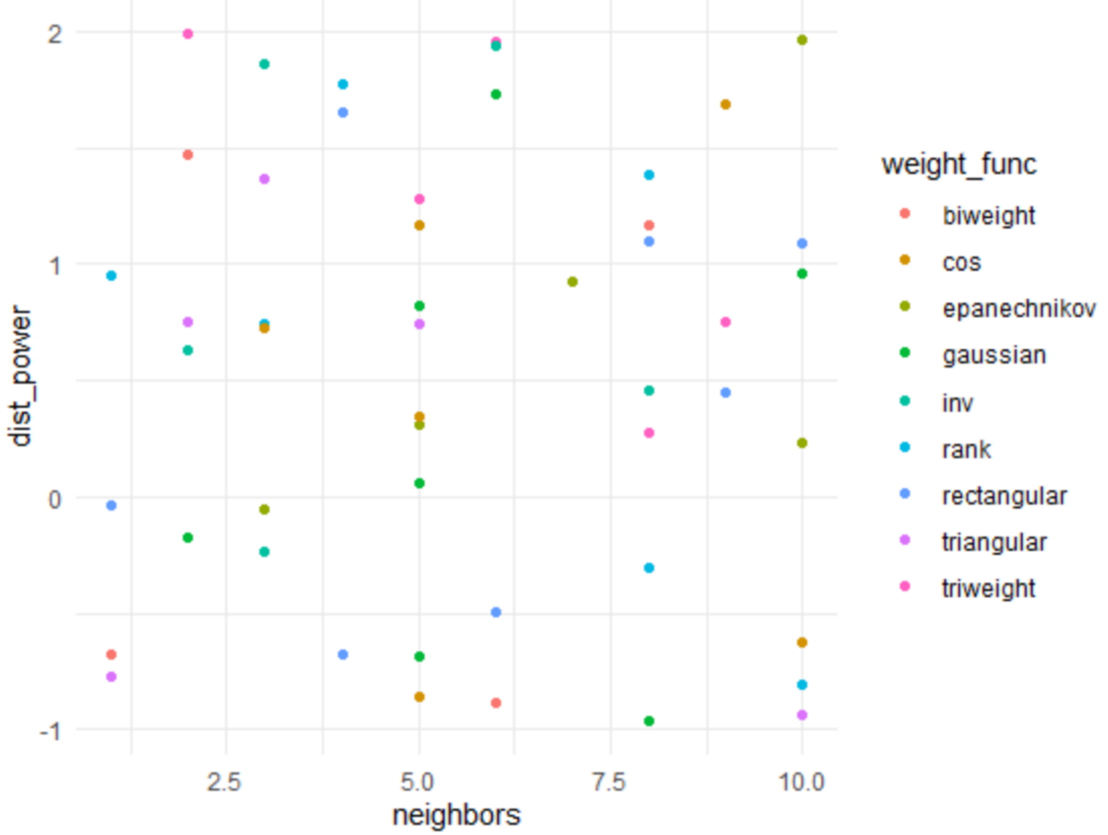


grid_max_entropy()

knn_params <- parameters(neighbors(), weight_func(), dist_power())

knn_sfd <- grid_max_entropy(knn_params, size = 50)

```
knn_sfd %>%
  ggplot(aes(neighbors, dist_power)) +
  geom_point(aes(color = weight_func))
```

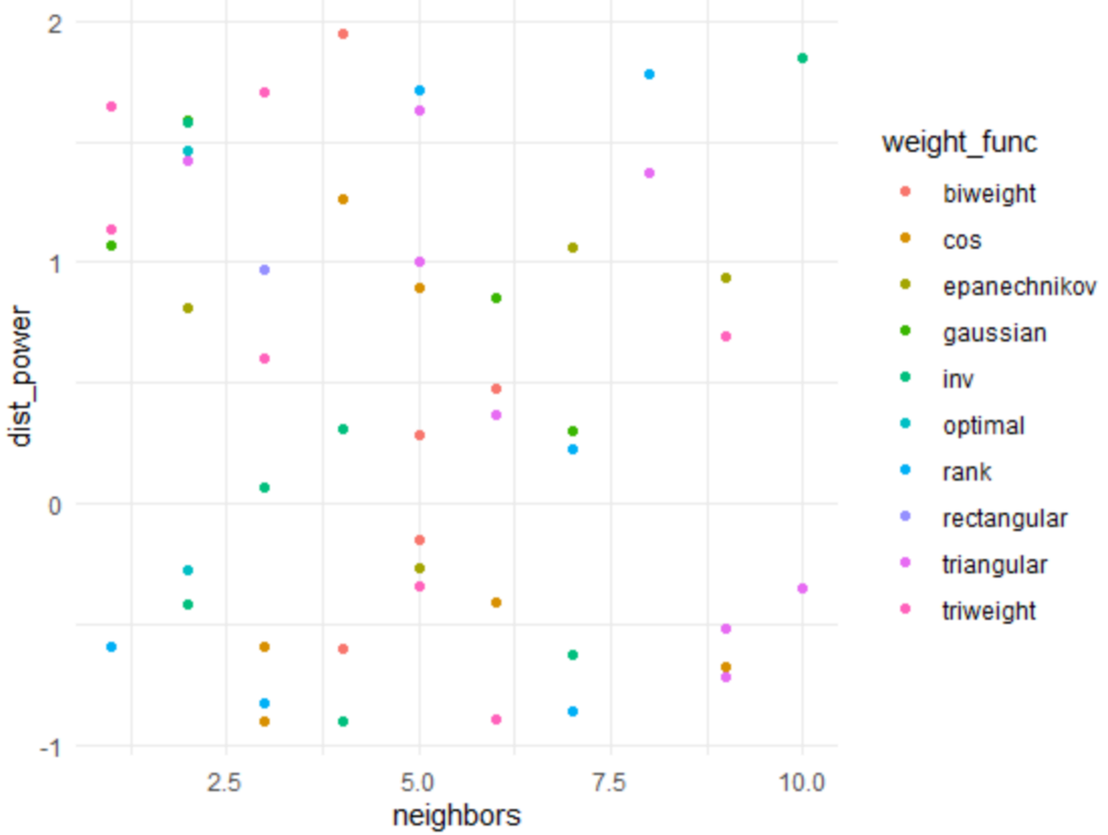


grid_random()

- Uniformly samples the parameter space without taking into account the previously generated sample points

knn_grid_ran <- grid_random(knn_params, size = 50)

```
knn_grid_ran %>%
  ggplot(aes(neighbors, dist_power)) +
  geom_point(aes(color = weight_func))
```

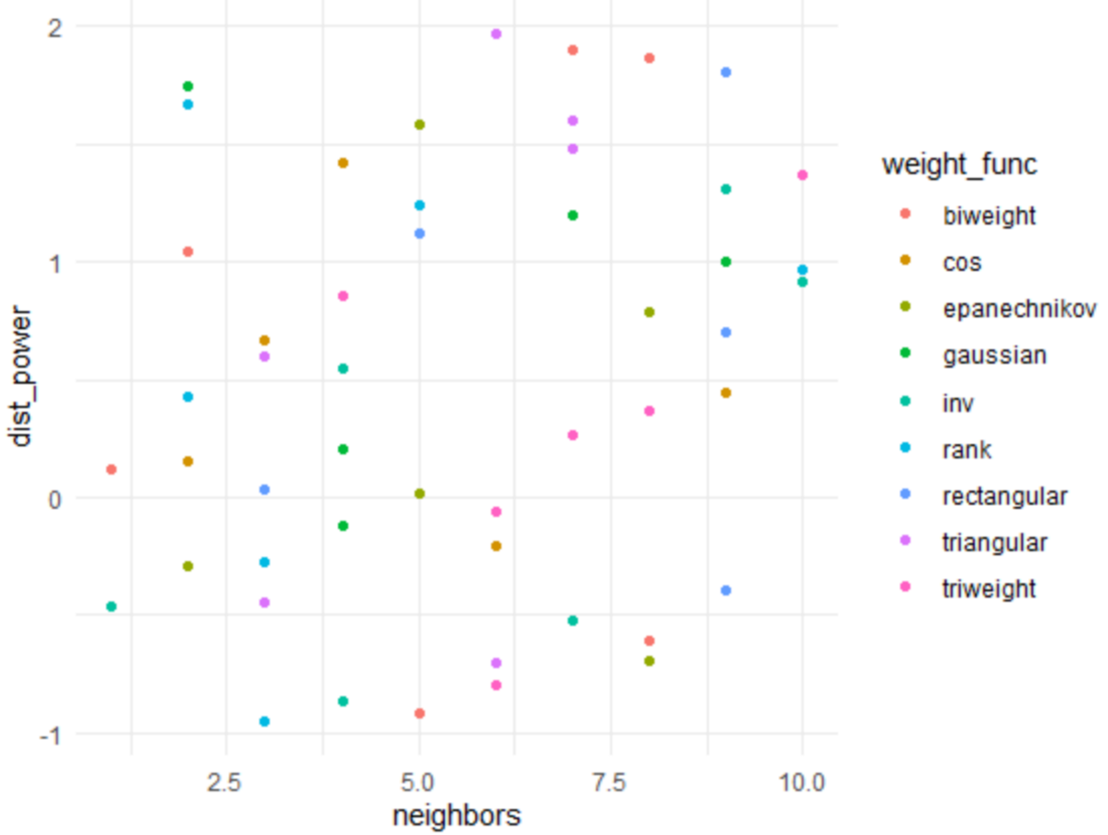


grid_latin_hypercube()

- Hyperspace generalization of a Latin square (one sample in each row and each column)

knn_grid_lhs <- grid_latin_hypercube(knn_params, size = 50)

```
knn_grid_lhs %>%
  ggplot(aes(neighbors, dist_power)) +
  geom_point(aes(color = weight_func))
```



Iterative searches

- grid searches
  - candidate values need to be pre-defined and don't learn from previous results
  - don't know the best values until all the computations are finished
  - difficult to efficiently cover the parameter space with a lot of parameters
  - easily optimized via parallel processing
- iterative searches
  - builds a probability model to predict better parameters to test based on previous results
  - more flexible in how the parameter space is searched
  - less opportunities for efficiency optimizations

List of iterative searches

- nonlinear search methods (computationally expensive)
  - Nelder-Mead simplex search procedure
  - simulated annealing
  - genetic algorithms
- Bayesian optimization
  - an initial pool of samples are evaluated using grid or random search
  - previous parameters used as predictors and performance measure used as the outcome
  - Bayesian optimization process searches the grid to find the "best" new parameters to evaluate using resampling
  - `{tune}` function is `tune_bayes()`

Let’s apply to a *K*NN model

- New recipe (adding predictors)

```
knn2_rec <-
  recipe(
    classification ~ enrl_grd + lat + lon + econ_dsvntg + sp_ed_fg,
    data = math_train) %>%
  step_mutate(classification = ifelse(classification < 3, "below", "proficient")) %>%
  step_mutate(enrl_grd = factor(enrl_grd)) %>%
  step_meanimpute(lat, lon) %>%
  step_string2factor(econ_dsvntg, sp_ed_fg) %>%
  step_unknown(enrl_grd, econ_dsvntg, sp_ed_fg) %>%
  step_dummy(enrl_grd, econ_dsvntg, sp_ed_fg) %>%
  step_normalize(lat, lon)
```

- New model

```
knn2_mod <- nearest_neighbor() %>%
  set_engine("kknn") %>%
  set_mode("classification") %>%
  set_args(neighbors = tune(),
           weight_func = tune())
```

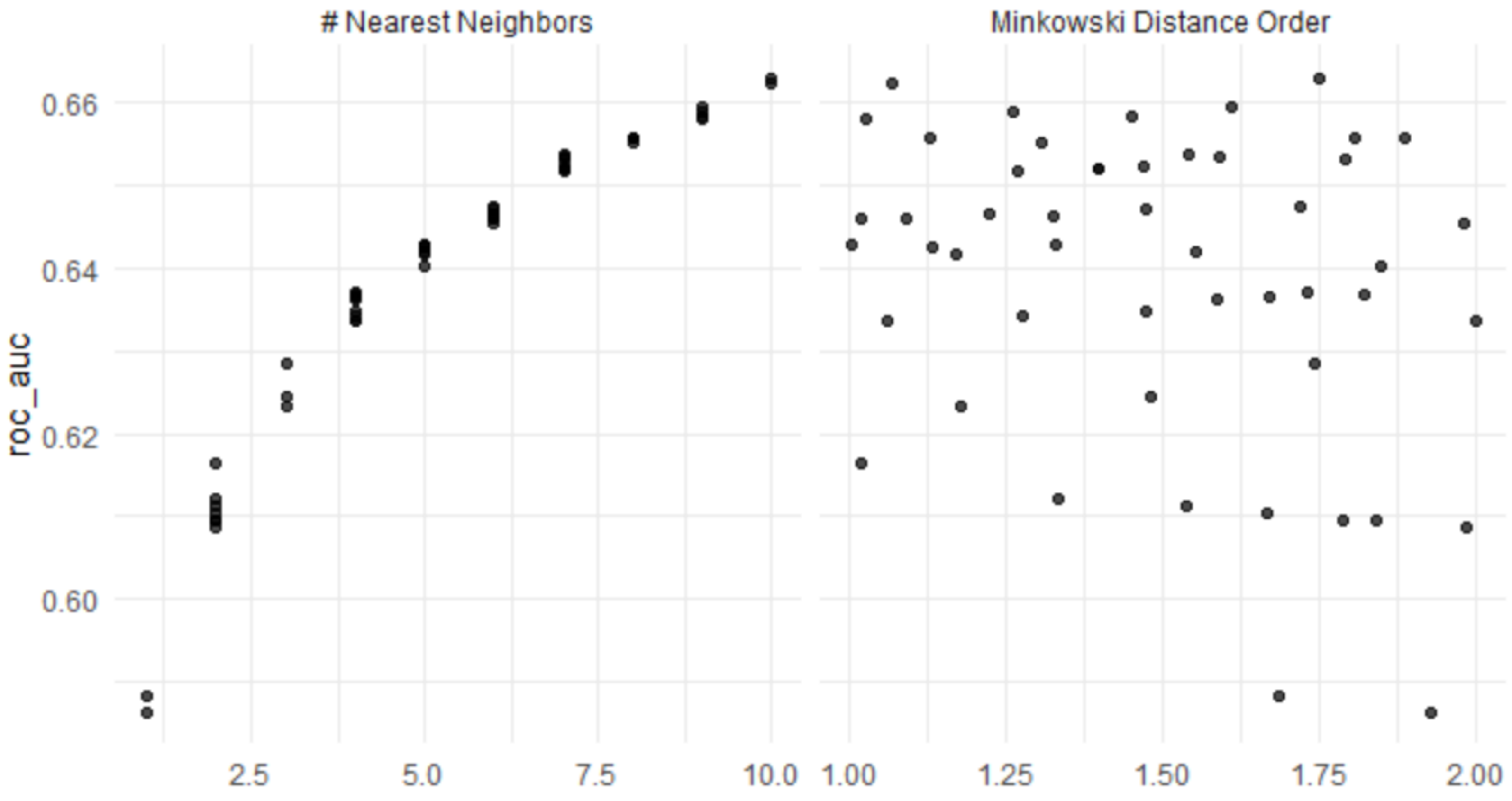
```r
# Let's make an SFD grid
knn_params <- parameters(neighbors(), dist_power())
knn_sfd <- grid_max_entropy(knn_params, size = 50)

# Tune
knn2_res <- tune::tune_grid(
  knn2_mod,
  preprocessor = knn1_rec,
  resamples = math_cv,
  grid = knn_sfd,
  control = tune::control_resamples(save_pred = TRUE)
)
```

```
knn2_res %>%
  collect_metrics()
```

```
# A tibble: 100 x 8
   neighbors dist_power .metric  .estimator  mean     n std_err .config
       <int>      <dbl> <chr>    <chr>      <dbl> <int>   <dbl> <chr>  
 1         5       1.00 accuracy binary     0.623    10  0.0111 Model01
 2         5       1.00 roc_auc  binary     0.643    10  0.0134 Model01
 3         2       1.02 accuracy binary     0.597    10  0.0121 Model02
 4         2       1.02 roc_auc  binary     0.616    10  0.0126 Model02
 5         6       1.02 accuracy binary     0.622    10  0.0104 Model03
 6         6       1.02 roc_auc  binary     0.646    10  0.0140 Model03
 7         9       1.03 accuracy binary     0.632    10  0.0115 Model04
 8         9       1.03 roc_auc  binary     0.658    10  0.0151 Model04
 9         4       1.06 accuracy binary     0.597    10  0.0116 Model05
10         4       1.06 roc_auc  binary     0.634    10  0.0127 Model05
# ... with 90 more rows
```

```
knn2_res %>%
  show_best(metric = "roc_auc", n = 5)

# A tibble: 5 x 8
  neighbors dist_power .metric .estimator  mean     n std_err .config
      <int>      <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>  
1        10       1.75 roc_auc binary     0.663    10  0.0154 Model39
2        10       1.07 roc_auc binary     0.662    10  0.0153 Model06
3         9       1.61 roc_auc binary     0.659    10  0.0158 Model32
4         9       1.26 roc_auc binary     0.659    10  0.0155 Model13
5         9       1.45 roc_auc binary     0.658    10  0.0157 Model22
```

```
knn2_res %>%
  autoplot(metric = "roc_auc")
```



- You could argue that the fit is improving and we should add more neighbors and explore

compare models

```
knn1_res %>%
  show_best(metric = "roc_auc", n = 1)
# A tibble: 1 x 9
  neighbors weight_func dist_power .metric .estimator  mean     n std_err .config
      <int> <chr>            <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
1        13 rank              1.23 roc_auc binary     0.585    10  0.0154 Model07
```

```
knn2_res %>%
  show_best(metric = "roc_auc", n = 1)
# A tibble: 1 x 8
  neighbors dist_power .metric .estimator  mean     n std_err .config
      <int>      <dbl> <chr>   <chr>      <dbl> <int>   <dbl> <chr>
1        10       1.75 roc_auc binary     0.663    10  0.0154 Model39
```

Final Fit

```
# Select best tuning parameters
knn_best <- knn2_res %>%
  select_best(metric = "roc_auc")

# Finalize your model using the best tuning parameters
knn_mod_final <- knn2_mod %>%
  finalize_model(knn_best)

# Finalize your recipe using the best turning parameters
knn_rec_final <- knn2_rec %>%
  finalize_recipe(knn_best)
```

Final Fit

```
# Run your last fit on your initial data split
cl <- makeCluster(8)
registerDoParallel(cl)
knn_final_res <- last_fit(
  knn_mod_final,
  preprocessor = knn_rec_final,
  split = math_split)
stopCluster(cl)

#Collect metrics
knn_final_res %>%
  collect_metrics()
# A tibble: 2 x 3
  .metric  .estimator .estimate
  <chr>    <chr>          <dbl>
1 accuracy binary         0.618
2 roc_auc  binary         0.651
```

Classification objective functions

```
knn_final_res %>%
  collect_predictions()

# A tibble: 947 x 6
   id               .pred_below .pred_proficient  .row .pred_class classification
   <chr>                  <dbl>            <dbl> <int> <fct>       <fct>
 1 train/test split      0.223             0.777     5 proficient  below
 2 train/test split      0.575             0.425     6 below       below
 3 train/test split      1                 0         7 below       below
 4 train/test split      0.873             0.127     8 below       proficient
 5 train/test split      0.244             0.756    18 proficient  below
 6 train/test split      0.311             0.689    19 proficient  proficient
 7 train/test split      0.640             0.360    27 below       below
 8 train/test split      0.0774            0.923    28 proficient  proficient
 9 train/test split      1                 0        31 below       below
10 train/test split      0.478             0.522    35 proficient  proficient
# ... with 937 more rows
```

- Columns 2 and 3 represent class probabilities for our two outcome classes
- The `.pred_class` column represents the class predicted by the model (class with highest probability)
  - Thus, most classification models can generate "hard" and "soft" predictions for models
  - The class predictions are usually created by thresholding some numeric output of the model (e.g. a class probability) or by choosing the largest value
- The `classification` column is the observed class (truth)

confusion matrix

```
knn_final_res %>%
  collect_predictions() %>%
  conf_mat(truth = classification, estimate = .pred_class)
```

```
            Truth
Prediction   below proficient
  below        379        181
  proficient   181        206
```

confusion matrix

```
knn_final_res %>%
  collect_predictions() %>%
  conf_mat(truth = classification, estimate = .pred_class)
```

```
            Truth
Prediction   below proficient
  below        379        181
  proficient   181        206
```

True Positive

confusion matrix

```
knn_final_res %>%
  collect_predictions() %>%
  conf_mat(truth = classification, estimate = .pred_class)
```

```
            Truth
Prediction   below proficient
  below        379        181
  proficient   181        206
```

True Negative

confusion matrix

```
knn_final_res %>%
  collect_predictions() %>%
  conf_mat(truth = classification, estimate = .pred_class)
```

```
              Truth
Prediction    below proficient
  below         379        181
  proficient    181        206
```

False Negative

confusion matrix

```
knn_final_res %>%
  collect_predictions() %>%
  conf_mat(truth = classification, estimate = .pred_class)
```

```
             Truth
Prediction    below proficient
  below         379        181
  proficient    181        206
```

False Positive

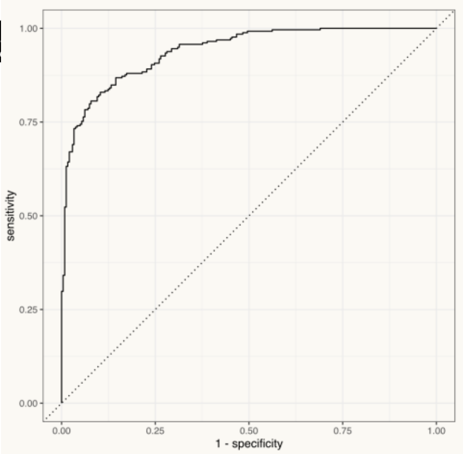
Classification objective functions

- conditional measures since we need to know the true outcome

`sens`: true positive rate; TP / (TP + FN)
- AKA: `recall`
- 1 - sensitivity = type-II error rate

`spec`: true negative rate: TN / (TN + FP)
- 1 – specificity = type-I error rate

`j_index`: `sens` + `spec` – 1
- Youden's J statistic

`roc_auc`: area under the curve receiver operatin 

urve

x-axis = 1 - `spec` (FPR)

y-axis = `sens` (TPR)

Classification objective functions

- `accuracy`: percent of outcomes correctly predicted; (TP + TN)/(TP+TN+FP+FN)
  - suffers when there is a class imbalance
- `kap`: Cohen's kappa, agreement adjusted for chance
- `ppv`: positive predictive vale; TP / (TP + FP)
  - AKA: `precision`
- `npv`: negative predictive value; TN / (FN + TN)
- `gain_capture`: area under gain curve and above the baseline, divided by area under a perfect gain curve and above the baseline
  - AKA: accuracy ratio (AR), gini coefficient

Which to use?

- Use the right criterion for your context
- Are true positives more valuable than true negatives?
  - Sensitivity will be important
- Do you want to have high confidence in predicted positives?
  - Precision will be important
- Are all errors equal?
  - Accuracy will work well
- There are a lot more!
  - `f_meas` combines precision and sensitivity

*K*NN for Imputation

Imputation

- Use information and relations among non-missing predictors to provide an estimate to fill in missing values
- KNN is also used in feature engineering to impute missing values
  - Primarily when the data is small-moderate in size
- Identifies the *K* (complete data) samples in the training data most similar to the missing value(s)
- The average value of the predictor of interest is calculated of the *K* closest samples and used to replace the missing value

Imputation

- When all predictors are numeric, standard **Euclidean distance** is commonly used as the similarity metric
- When predictors are numeric and categorical, **Gower's distance** is recommended (Kuhn & Johnson, 2019)
  - Categorical: the distance is 1 if the samples have the same value and 0 if not
  - Numeric: $d(x_i, x_j) = 1 - \frac{|x_i - x_j|}{R_x}$, where R_x is the range of the predictor *x*
- *K* is a tunable parameter, but values around 5–10 are a good default