Supervised Learning

K-Nearest Neighbors Regression and Classification

June 30th, 2023

Supervised learning so far...

Linear Regression: Assumptions

- The relationship is **linear**

$$Y_i = \beta_0 + \beta_1 X_i + \epsilon_i, \quad \text{for } i = 1, 2, \ldots, n$$

$$\epsilon_i \overset{iid}{\sim} N(0, \sigma^2) \quad \text{with constant variance } \sigma^2$$

- Error terms must be **independently, identically distributed** from Normal distribution (actually three assumptions!)
  - Normality
  - Independence
  - Homoscedasticity

Each of these assumptions must be **checked**

Assumptions (in general) make estimation and inference easier but may not always be appropriate

Parametric models

When we make assumptions about distributions, we are saying that we can summarize them with a set of **parameters**

Examples...

- Normal distribution: $N(\mu, \sigma^2)$, the **mean** μ and **variance** σ^2 are parameters
- Poisson distribution: $\text{Pois}(\lambda)$, the only parameter is λ, which is both the **mean and variance**
- Simple linear regression: $Y = \beta_0 + \beta_1 X$, the coefficients (**intercept** β_0 and **slope** β_1) are parameters

But what if we don't want to make assumptions **a priori** about our data?

Nonparametric models

Still assume a function mapping the predictors to the response

$$Y = f(X)$$

And we still want to **estimate** this regression or classification function

But now we do not assume that this function takes any **particular form** (e.g. do not assume that the relationship is linear)

Benefits

- Very flexible, allowing for fitting to very complex relationships
- Often show better performance, more accurate predictions

Limitations

- Frequently less interpretable
- Prone to overfitting
- Computational concerns (require more data, take longer to train)

Regression vs. classification

- **Regression** models: estimate *average* value of response
- **Classification** models: determine *the most likely* class of a set of discrete response variable classes

Linear (simple or multiple) models $\rightarrow$ regression

Logistic models $\rightarrow$ classification!

Examples of classification:

- Binary: given covariates like running yards and completion percentage, can we predict **whether or not** a QB is likely to be a Hall-of-Famer?
- Multi-class: from predictors like tumor size, location, and cell types, can we determine **which kind** of cancer a patient has?

Question of interest determines which type of model to use

Many nonparametric methods (e.g. KNN) can be used either for classification **OR** regression, just change the final outcome

Setup: Binary classification problem

$Y = f(X_1, X_2)$

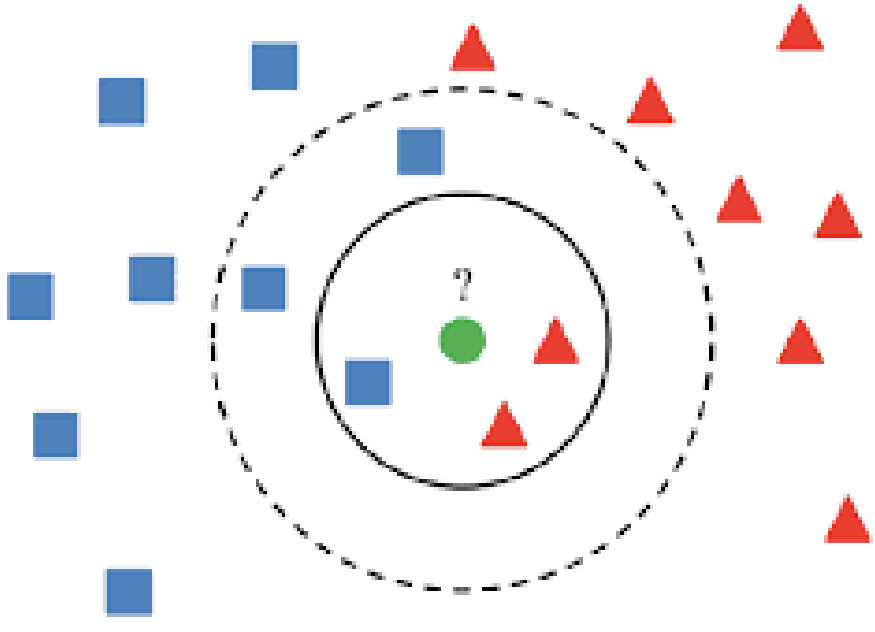
$Y \in \{0, 1\}$

How might we model $f(X_1, X_2)$?

"Decision boundary"

Example settings:

- QB's running and passing yards (X_1, X_2), HOF status (Y)
- Patient's length of inpatient rehab stay and report of outpatient stress (X_1, X_2), substance abuse relapse (Y)



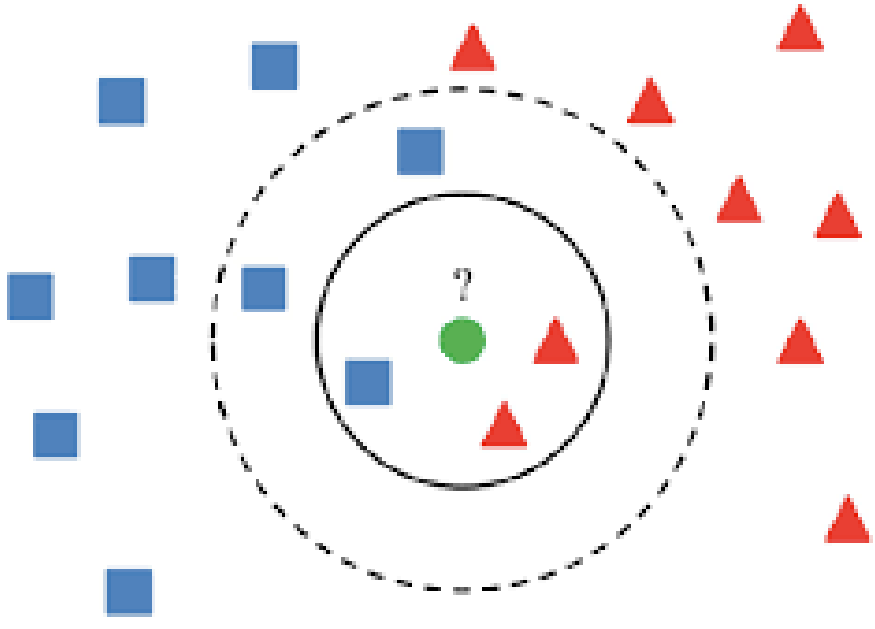
K-Nearest Neighbors Classification

Classify a **new point** based on a majority vote of the **k** points closest to it in space

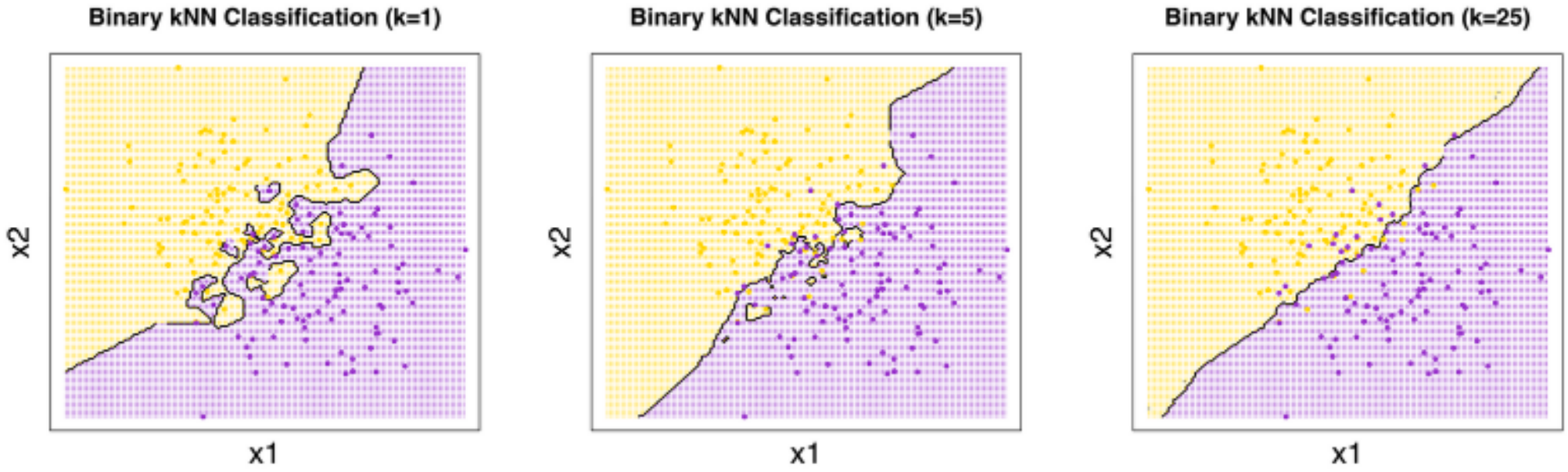
Close? Use Euclidean distance

$$d(x_i, x_j) = \sqrt{(x_{i1} - x_{j1})^2 + \cdots + (x_{ip} - x_{jp})^2}$$

Have to choose how many neighbors to query: which value of **k** to pick



Different values of k lead to different decision boundaries



Excellent illustration of the **bias-variance tradeoff**

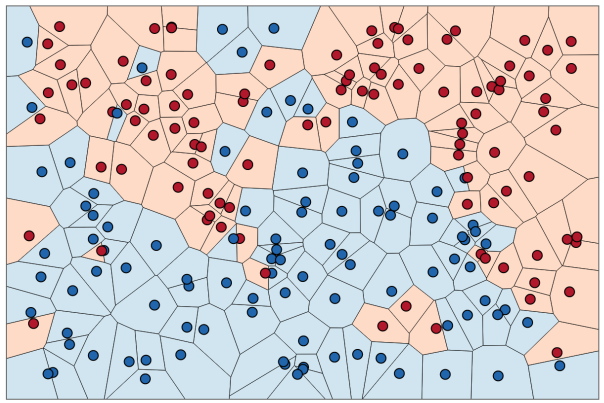
As **k increases**, the **variance decreases** (different training samples will produce more similar decision boundaries), but the **bias increases** (the decision boundary gets further from the truth)

Picking the number of neighbors

The number of neighbors **k** is a **hyperparameter**

Tune this value using a train-test split or using cross-validation

Note: 1NN classification will have zero training error... **why** ?



Ideally: balance the flexibility, allowing for a complex decision boundary, but don't fit to the noise

Data: NFL field goal attempts

Created dataset using `nflscrapR-data` of all NFL field goal attempts from 2009 to 2019

```
nfl_fg_attempts <- read_csv("https://shorturl.at/mCGN2") %>%
  filter(pbp_season == 2014) %>%
  mutate(is_fg_made = as_factor(is_fg_made))
head(nfl_fg_attempts)
```

```
## # A tibble: 6 × 11
##   kicker_player_id kicker_player_name   qtr score_differential home_team posteam
##   <chr>            <chr>              <dbl>              <dbl> <chr>     <chr>
## 1 00-0025944       S.Hauschka             1                  0 SEA       SEA
## 2 00-0025580       M.Crosby               2                 -3 SEA       GB
## 3 00-0025944       S.Hauschka             3                  7 SEA       SEA
## 4 00-0019536       S.Graham               1                  0 ATL       NO
## 5 00-0019536       S.Graham               1                  3 ATL       NO
## 6 00-0020578       M.Bryant               2                -13 ATL       ATL
## # ℹ 5 more variables: posteam_type <chr>, kick_distance <dbl>,
## #   pbp_season <dbl>, abs_score_diff <dbl>, is_fg_made <fct>
```

Response: is the field goal made?

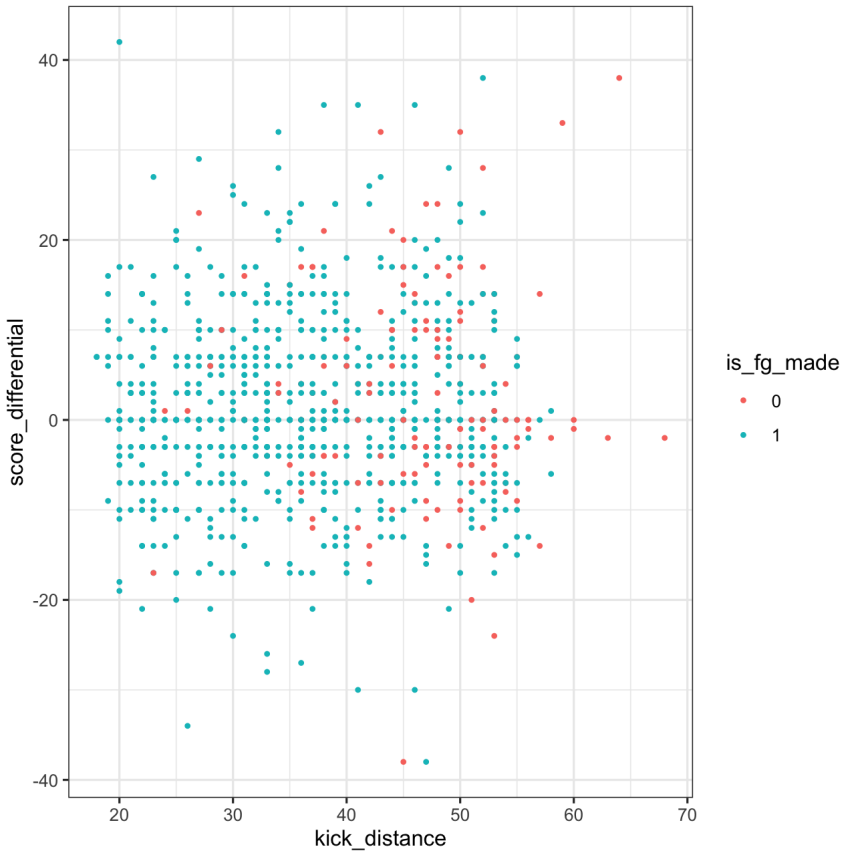
Predicting field goals

Explanatory variables:

Kick distance and score differential

```
ggplot(nfl_fg_attempts) +
  geom_point(aes(x = kick_distance,
                 y = score_differential,
                 color = is_fg_made),
             size = 0.7) +
  theme_bw()
```

Does there appear to be a relationship between these predictors and the success of field goal attempts?



KNN classification

Using the FNN package (fast implementation of KNN models)

Separate predictors from the response column:

```
fg_x <- dplyr::select(nfl_fg_attempts, kick_distance, score_differential)
fg_y <- nfl_fg_attempts$is_fg_made
```

Give `knn()`

- a training set with all explanatory columns `train`
- a "test" set including all explanatory columns `test`
  - if you provide the same set for both, it will give predictions on the training set
- a vector of true classifications for the training set `cl`
- a value for number of nearest neighbors `k`
- instructions for which `algorithm` to use ("brute" means that it will do a brute-force search, but it can also do tree-based searches)

Fitting a KNN classifier

Trying $k = 1$:

```
library(FNN)

init_knn <- knn(train = fg_x, test = fg_x, cl = fg_y, k = 1, algorithm = "brute")
```

This outputs a vector of predicted classes for the "test" set (in this case, predictions on the training set)

How well does this model perform?

```
mean(nfl_fg_attempts$is_fg_made == init_knn)
```

```
## [1] 0.9177665
```

92% But we said that 1NN would achieve zero training error?

In this dataset, there are multiple observations for some distance-differential combinations, which might have both made and missed field goals, so it then takes the majority vote among those "equivalent" points (see the documentation!)

Training vs. test

But, as we know, we should not assess models based on training error, we need some measure of holdout performance:

Split the data, train the model, then assess performance on the test set

(Yet another way to manually code a train-test split)

```
set.seed(50)
train_ids <- sample(1:nrow(nfl_fg_attempts),
                    ceiling(0.75 * nrow(nfl_fg_attempts)))

train_nfl <- nfl_fg_attempts[train_ids, ]
test_nfl <- nfl_fg_attempts[-train_ids, ]

# separate predictors and response, again
train_x <- dplyr::select(train_nfl, kick_distance, score_differential)
train_y <- train_nfl$is_fg_made
test_x <- dplyr::select(test_nfl, kick_distance, score_differential)
test_y <- test_nfl$is_fg_made
```

Assessing 1NN

1 Nearest Neighbor classifier: How well does it perform on the training data?

```
one_nn_train_preds <- knn(train = train_x, test = train_x, cl = train_y, k = 1, algorithm = "brut
mean(train_y == one_nn_train_preds)
```

```
## [1] 0.9418133
```

... But what about on the test data?

```
one_nn_test_preds <- knn(train = train_x, test = test_x, cl = train_y, k = 1, algorithm = "brute'
mean(test_y == one_nn_test_preds)
```

```
## [1] 0.7479675
```

Performance decreases! We're overfitting!

Which value of k to pick?

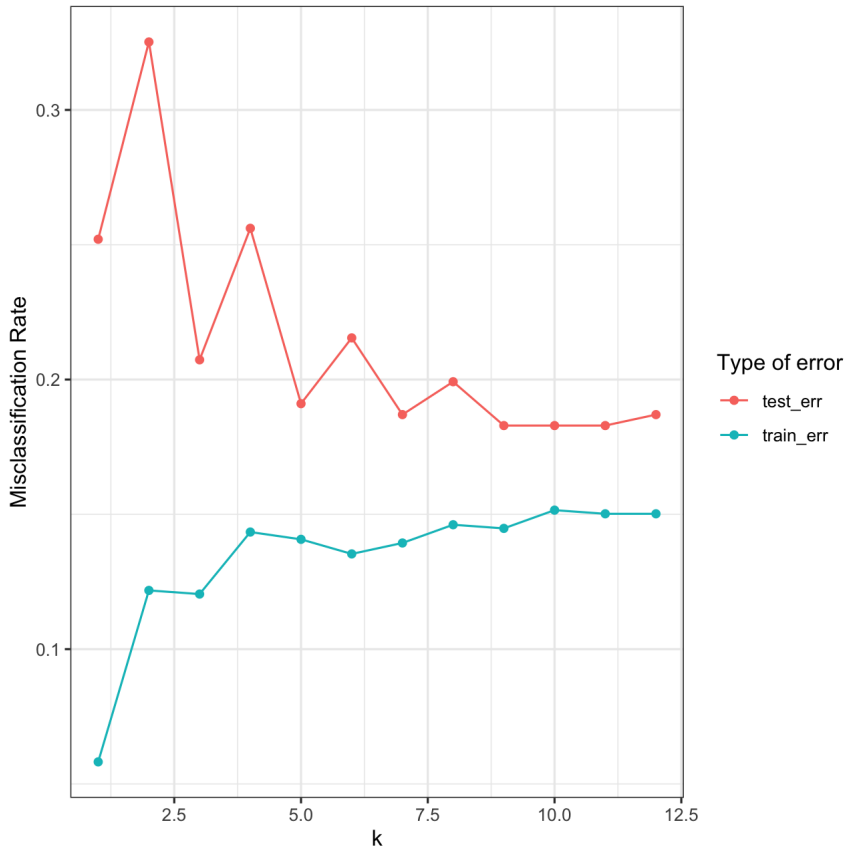
As a demonstration, let's loop through possible values of k and see which ends up having the best holdout performance

```
errs_train <- rep(0, 12)
errs_test <- rep(0, 12)
k_vals <- 1:12

for(k in k_vals) {

  train_preds <- knn(train = train_x, test = train_x, cl = train_y, k = k, algorithm = "brute")
  errs_train[k] <- mean(!train_y == train_preds)

  test_preds <- knn(train = train_x, test = test_x, cl = train_y, k = k, algorithm = "brute")
  errs_test[k] <- mean(!test_y == test_preds)
}
```

Side note: beyond k = 12, the model predicts success for the entire test set

Plot training and test error

```
errors <- bind_cols(train_err = errs_train,
                    test_err = errs_test,
                    k = k_vals)

errors %>%
  pivot_longer(c(train_err, test_err),
               names_to = "err_type") %>%
  ggplot(aes(x = k, y = value,
             color = err_type)) +
  geom_point() +
  geom_line() +
  labs(y = "Misclassification Rate",
       color = "Type of error") +
  theme_bw()
```



But wait, there's more!

The k-nearest-neighbors algorithm can also be used for **regression**!

As before:

- Find the k points closest to your new observation (based on Euclidean distance)

But now,

- Rather than taking a *majority vote* to determine a class, take the **average response** among the neighbors
- Predict this average value as the response for the new point:

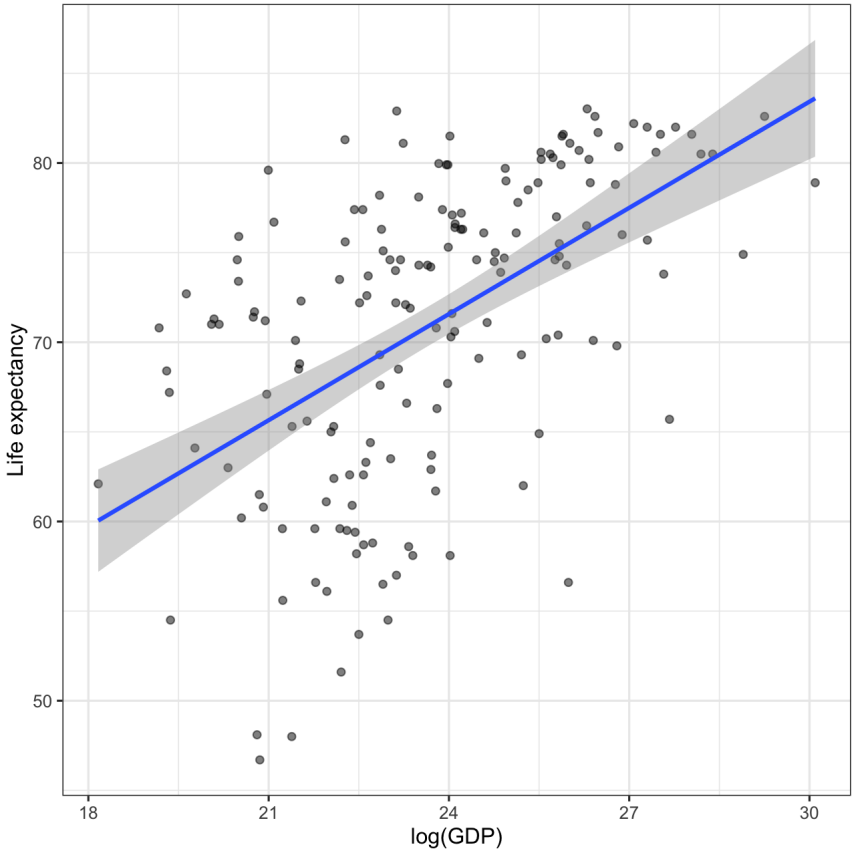
$$\hat{f}(x^*) = \text{Ave}(y_i | x_i \in N_k(x^*))$$

(**Side Note**: You can also do cool stuff like weighting those responses based on how close they are to the new point! Lots of modifications to knn regression out there)

Brief KNN Regression Example

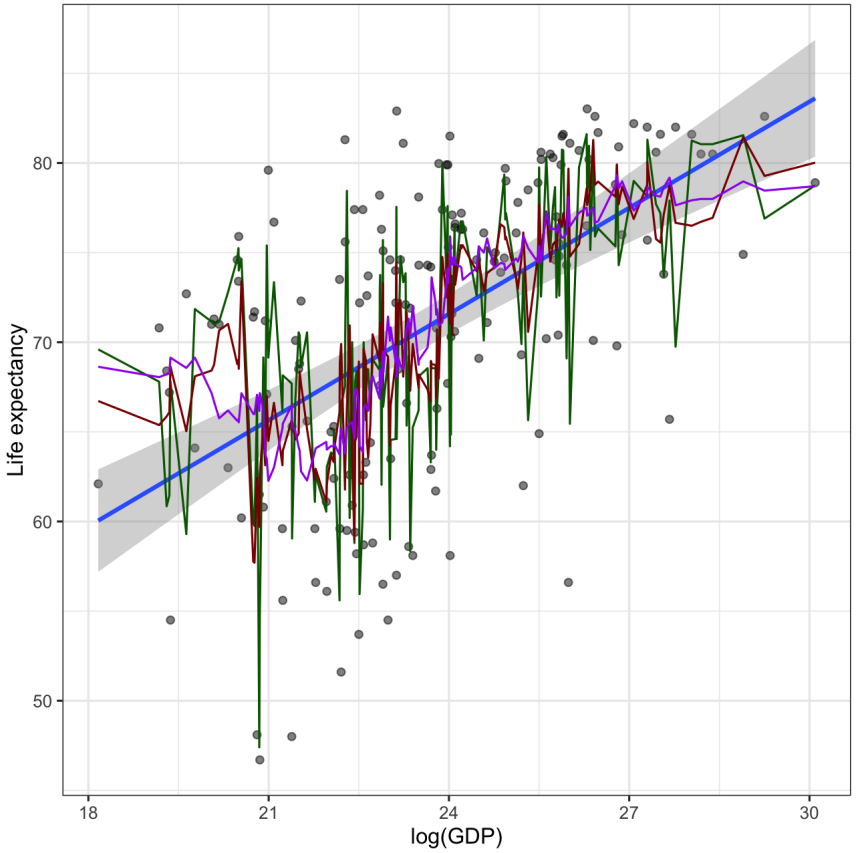
Gapminder data again

```
library(dslabs)
clean_gapminder <- as_tibble(gapminder) %>%
  filter(year == 2011, !is.na(gdp)) %>%
  mutate(log_gdp = log(gdp))

gdp_plot <- clean_gapminder %>%
  ggplot(aes(x = log_gdp,
             y = life_expectancy)) +
  geom_point(alpha = 0.5) +
  geom_smooth(method = "lm") +
  theme_bw() +
  labs(x = "log(GDP)",
       y = "Life expectancy")
gdp_plot
```



KNN Regression for Life Expectancy

```
k2_le <- knn.reg(
  train = clean_gapminder$log_gdp,
  y = clean_gapminder$life_expectancy,
  k = 2)
k5_le <- knn.reg(
  train = clean_gapminder$log_gdp,
  y = clean_gapminder$life_expectancy,
  k = 5)
k15_le <- knn.reg(
  train = clean_gapminder$log_gdp,
  y = clean_gapminder$life_expectancy,
  k = 15)

gdp_plot +
  geom_line(aes(x = log_gdp, y = k2_le$pred),
            color = "darkgreen") +
  geom_line(aes(x = log_gdp, y = k5_le$pred),
            color = "darkred") +
  geom_line(aes(x = log_gdp, y = k15_le$pred)
            color = "purple")
```



KNN with tidymodels

Of course, we can also use the tidymodels framework to fit and use a KNN regression model:

```
library(tidymodels)
knn_reg_mod <- nearest_neighbor(
  mode = "regression",
  engine = "kknn",
  neighbors = 5,          # we could also tune this with cross-validation
  weight_func = NULL,     # for weighted KNN!
  dist_power = NULL       # for use with a different distance metric
)

gapminder_knn_fit <- knn_reg_mod %>%
  fit(
    life_expectancy ~ log_gdp,
    data = clean_gapminder
  )
```