…There are a variety of ways to go about explaining model features, but probably the most common approach is to use **variable (or feature) importance** scores. Unfortunately, computing variable importance scores isn't as straightforward as one might hope—there are a variety of methodologies! Upon implementation, I came to the question "How similar are the variable importance scores calculated using different methodologies?" [1] I think it's important to know if the different methods will lead to drastically different results. If so, then the choice of method is a source of bias in model interpretation, which is not ideal.

This post isn't intended to be a deep-dive into model interpretability or variable importance, but some concepts should be highlighted before attempting to answer this question. Generally, variable importance can be categorized as either being "model-specific" or "model-agnostic". Both depend upon some kind of loss function, e.g. root mean squared error (RMSE), classification error, etc. The loss function for a model-specific approach will generally be "fixed" by the software and package that are used [2], while model-agnostic approaches tend to give the user flexibility in choosing a loss function. Finally, within model-agnostic approaches, there are different methods, e.g. permutation and SHAP (Shapley Additive Explanations).

So, to summarize, variable importance "methodologies" can be broken down in several ways:

1. model-specific vs. model-agnostic approach
2. loss function
3. model agnostic method (given a model agnostic approach)

I'm going to attempt to address (1) and (3) above. I'm leaving (2) out because (a) I think the results won't differ too much when using different loss functions (although I haven't verified this assumption) and (b) for the sake of simplicity, I don't want to be too exhaustive in this analysis. [3]

I also want to evaluate how variable importance scores differ across more than one of each of the following:

1. model type (e.g. linear regression, decision trees, etc.)
2. type of target variables (continuous or discrete)
3. data set

While evaluating the sensitivity of variable importance scores to different methodologies is the focus of this analysis, I think it's important to test how the findings hold up when (1) varying model types, (2) varying target variables, and (3) varying the data itself. This should help us highlight any kind of bias in the results due to choice of model type and type of target variable. Put another way, it should help us quantify the robustness the conclusions that are drawn. If we find that the scores are similar under variation, then we can be more confident that the findings can be generalized.

Additionally, I'm going to use more than one package for computing variable importance scores. As with varying model types, outcome variables, and data, the purpose is to highlight and quantify possible bias due to choices in this analysis—in this case, the choice of package. Are the results of a permutation-based variable importance calculation the same when using different packages (holding all else equal)?

Specifically, I'll be using the `{vip}` and `{DALEX}` packages. The `{vip}` package is my favorite package to compute variable importance scores using R is because it is capable of doing both types of calculations (model-specific and model-agnostic) for a variety of model types. But other packages are also great. `{DALEX}` package specializes in model-agnostic model interpretability and can do a lot more than just variable importance calculations.

Setup

For data, I'm going to be using two data sets [4]:

1. `diamonds` from `{ggplot2}`. [5]
2. `mpg` from `{ggplot2}`. [6]

I made modifications to both, so see the footnotes and/or code if you're interested in the details.

For model types, I’m going to trial the following:

1. generalized linear model (linear and logistic regression) with `stats::lm()` and `stats::glm()` respectively
2. generalized linear model with regularization using the `{glmnet}` package
3. bagged tree (random forest) using the `{ranger}` package
4. boosted tree (extreme gradient boosting) using the `{xgboost}` package

With `glmnet::glmnet()`, I’m actually not going to use a penalty, so (I think) it should return the same results as `lm()`/`glm()`.[7] For `{ranger}` and `{xgboost}`, I’m going to be using defaults for all parameters.[8]

1. `{vip}`’s model-specific scores with (`vip::vip(method = 'model')`)
2. `{vip}`’s permutation-based scores (with `vip::vip(method = 'permute')`)
3. `{vip}`’s SHAP-based values (with `vip::vip(method = 'shap')`)
4. `{DALEX}`’s permutation-based scores (with `DALEX::variable_importance()`)

Note that the model-specific vs. model-agnostic concern is addressed in comparing method (1) vs. methods (2)-(4). I’ll be consistent with the loss function in variable importance computations for the model-agnostic methods–minimization of RMSE for a continuous target variable and sum of squared errors (SSE) for a discrete target variable.[9]

Results

The following handful of plots illustrate normalized variable importance scores and ranks derived from the scores by data set and type of target variable.

First up is the results for the `diamonds` data set with a continuous target variable.

One thing really stand out to me: the model-specific scores differ relatively strongly from the rest of the scores given a specific model type. (See the numbers in the parentheses in the first column in each facet labeled `vip_model` compared to those in the other columns of each facet.[10] For example, the model-specific variable importance score for the `carat` feature for the `{glm}` model type is 49%, while the same score for the SHAP variable importance method (`vip_shap`) is 35%. To be honest, this is not too surprising. The model-specific methods are exactly that—specific to the model type—which suggests that they may strongly dissimilar to the model-agnostic approaches. Nonetheless, despite the scores themselves having some notable variance, the rankings derived from the scores are relatively similar across a given model type (and, arguably, across all model types).

As a second observation, there is some disagreement between the `{glm}` and `{glmnet}` model types and the `{ranger}` and `{xgboost}` model types about which feature is the most important: the former two identify `carat` has being the most important, while the latter two prioritize `y`.

Thirdly–and lastly for this plot—it’s nice to see that the `vip_permute` and `dalex` methods produce nearly identical results for each model type, with the exception of `{glmnet}`. (Hypothetically, these should have nearly identical results since they are both permutation based methods.) Notably, I implemented the `explain()` function for `{glmnet}` myself since the `{DALEX}` package does not export one, so that is probably the reason for the discrepancy 😄.

Now let’s look at the the results when predicting a discrete target variable with the same data set.

Compared to the results for a continuous target variable, we see greater variation across the model types—the rankings from `{glm}` and `{glmnet}` are nearly identical, but they are different from those of `{xgboost}`, and all are different from those of `{ranger}`. `{ranger}` has an additional level of variation—

lack of agreement among the methodologies.

Additionally, we observe that the scores for our two permutation implementations— `vip_permute` and `dalex`—are **very** different. I think this might have to do with how I've chosen to normalize scores (i.e. using absolute value to convert negative scores to positive ones prior to 0-1 normalization) or something I've overlooked that is specific to classification settings. If something that can be attributed to me (and not the underlying methods) is really the source of discrepancies, then we should be less concerned with the variation in scores and ranks since it seems most strongly associated with the `vip_permute`–`dalex` differences.

Before we can begin to generalize any deductions (possibly biased by our single data set), let's take a look at the results for the second data set, `mpg`. First is the results for the continuous target variable.

There is consensus on what the most important variable is—`cyl`—but beyond that, the results are somewhat varied across the board. One might argue that there is going to be lack of agreement among methods (and model types), it's preferable that the discrepancies occur among lower ranks, as seen here. On the other hand, we'd surely like to see more consensus among variables ranked among the top half or so.

And now for the results when ranking with models targeting a discrete variable.

There is some pretty strong variation in the `{ranger}` results. Also, there are discrepancies between the two permutation methods (`vip_permute` and `dalex`), which we also noted in the discrete results for `diamonds` as well. This makes me think again that the issue is due to something I've done and not something that could be attributed to the underlying methods. Aside from these, I would say that the results within each model type are pretty coherent (more so than those with the continuous outcome.)

Even without performing any kind of similarity evaluation, we can argue that, in general, the rankings computed by the different methods are relatively similar across the two data sets (`diamonds` and `mpg`) and the two types of target variables (continuous and discrete). But why stop there? After all, we **can** quantify the similarities between ranks.

The plot above shows the pairwise correlations among the variable importance ranks computed for each package-function combo, averaged over the two data sets and over the models for the two types of target variables—continuous and discrete. [11] While nothing immediately jumps out from this plot, I think the most notable thing is that the `{ranger}` scores seem to vary the most across the different variable importance methodologies, bottoming out at ~74% for the correlation between the SHAP (`vip_shap`) and model-specific (`vip_model`) methodologies. On the other hand, `{xgboost}` seems to have the most "agreement" and least variance in its scores.

Conclusion

Overall, we might say that rankings of variable importance based on normalized variable importance scores in this analysis showed that differences will arise when evaluating different methodologies, but the differences may not be strong enough to change any deductions that one might draw. Of course, this will depend on the context. A small differences could make a huge difference in a field like medicine!

I wouldn't go so far as to say that these insights can be generalized—among other things, I think I would need to evaluate a much larger variety of data sets—but I think it's good to be conscious how much the results can vary. It's ultimately up to the user whether the differences are significant.

Appendix

See all relevant R code below.

```
library(tidyverse)
.seed <- 42L # Also using this immediately before data set splitting with
`{rsample}`.
set.seed(.seed)
diamonds_modified <-
  ggplot2::diamonds %>%
  sample_frac(0.05) %>%
  mutate(
    color =
      case_when(
        color %in% c('D', 'E') ~ 'DE',
        color %in% c('F', 'G') ~ 'FG',
        TRUE ~ 'HIJ'
      ) %>% as.factor()
  ) %>%
  mutate(
    grp =
      case_when(
        cut %in% c('Idea', 'Premium') ~ '1. Best',
        TRUE ~ '2. Worst'
      ) %>% as.factor()
  ) %>%
  select(-cut, -clarity)
diamonds_modified

mpg_modified <-
  ggplot2::mpg %>%
  mutate(
    grp =
      case_when(
        class %in% c('2seater', 'compact', 'subcompact', 'midsize') ~ '1.
Small',
        TRUE ~ '2. Big')
  ) %>%
  select(-class, -model, -manufacturer, -trans, -fl)
mpg_modified

explain.glmnet <-
  function (object,
            feature_names = NULL,
            X,
            nsim = 1,
            pred_wrapper,
            newdata = NULL,
            exact = FALSE,
            ...) {

    if (isTRUE(exact)) {
      if (is.null(X) && is.null(newdata)) {
        stop('Must supply `X` or `newdata` argument (but not both).', call. =
FALSE)
      }
      X <- if (is.null(X))
        newdata
      else X
      res_init <- stats::predict(object, newx = X, s = 0, type = 'coefficients',
...)

      # https://stackoverflow.com/questions/37963904/what-does-predict-glm-type-terms-actually-do
      beta <- object %>% coef(s = 0) %>% as.matrix() %>% t()
      avx <- colMeans(X)
      X1 <- sweep(X, 2L, avx)
      res <- t(beta[-1] * t(X1)) %>% as_tibble() %>% mutate_all(~coalesce(., 0))
      attr(res, which = 'baseline') <- beta[[1]]
      class(res) <- c(class(res), 'explain')
      res
    } else {
      fastshap::explain.default(
        object,
        feature_names = feature_names,
        X = X,
        nsim = nsim,
        pred_wrapper = pred_wrapper,
        newdata = newdata,
        ...
      )
    }
  }

vip_wrapper <- function(method, ...) {
  res <-
    vip::vip(
      method = method,
      ...
    ) %>%
    pluck('data') %>%
    # Will get a "Sign" solumn when using the default `method = 'model'`.
    rename(var = Variable, imp = Importance)

  if(any(names(res) == 'Sign')) {
    res <-
      res %>%
      mutate(dir = ifelse(Sign == 'POS', +1L, -1L)) %>%
      mutate(imp = dir * imp)
  }
  res
}

# 'glm' gets converted to 'lm' for regression in my code
.engines_valid <- c('glm', 'glmnet', 'xgboost', 'ranger')
engines_named <- .engines_valid %>% setNames(., .)
.modes_valid <- c('regression', 'classification')
choose_f_fit <- function(engine = .engines_valid, mode = .modes_valid) {
  engine <- match.arg(engine)
  mode <- match.arg(mode)
  f_glm <- list(parsnip::linear_reg, parsnip::logistic_reg) %>%
set_names(.modes_valid)
  fs <-
    list(
      'xgboost' = rep(list(parsnip::boost_tree), 2) %>% set_names(.modes_valid),
      'ranger' = rep(list(parsnip::rand_forest), 2) %>% set_names(.modes_valid),
      'glm' = f_glm,
      'glmnet' = f_glm
```

```
    )
  res <- fs[[engine]][[mode]]
  res
}

choose_f_predict <- function(engine = .engines_valid) {
  engine <- match.arg(engine)
  f_generic <- function(object, newdata) predict(object, newdata = newdata)
  fs <-
    list(
      'xgboost' = f_generic,
      'ranger' = function(object, newdata) predict(object, data =
newdata)$predictions,
      'glm' = f_generic,
      # Choosing no penalty.
      'glmnet' = function(object, newdata) predict(object, newx = newdata, s =
0)
    )
  fs[[engine]]
}

is_binary <- function(x) {
  n <- unique(x)
  length(n) - sum(is.na(n)) == 2L
}

is_discrete <- function(x) {
  is.factor(x) | is.character(x)
}

# I would certainly not recommend a big function like this in a normal type of
project or analysis. But, in this case, it makes things more straightforward.
compare_and_rank_vip <-
  function(data,
           col_y,
           engine = .engines_valid,
           mode = NULL,
           ...) {
    message(engine)
    engine <- match.arg(engine)

    if(!is.null(mode)) {
      mode <- match.arg(mode, .modes_valid)
    } else {
      y <- data[[col_y]]
      y_is_discrete <- is_discrete(y)
      y_is_binary <- is_binary(y)

      mode <-
        case_when(
          y_is_discrete | y_is_binary ~ 'classification',
          TRUE ~ 'regression'
        )
    }

    mode_is_class <- mode == 'classification'
    parsnip_engine <-
```

```
      case_when(
        engine == 'glm' & !mode_is_class ~ 'lm',
        TRUE ~ engine
      )

    f_fit <- choose_f_fit(engine = engine, mode = mode)
    fmla <- formula(sprintf('%s ~ .', col_y))
    set.seed(.seed)
    splits <- data %>% rsample::initial_split(strata = col_y)

    data_trn <- splits %>% rsample::training()
    data_tst <- splits %>% rsample::testing()

    rec <-
      recipes::recipe(fmla, data = data_trn) %>%
      # Note that one-hot encoding caused rank deficiencies with `glm()` and
`{DALEX}`.
      recipes::step_dummy(recipes::all_nominal(), -recipes::all_outcomes(),
one_hot = FALSE)

    is_ranger <- engine == 'ranger'
    f_engine <- if(is_ranger) {
      partial(parsnip::set_engine, engine = parsnip_engine, importance =
'permutation')
    } else {
      partial(parsnip::set_engine, engine = parsnip_engine)
    }

    spec <-
      f_fit() %>%
      f_engine() %>%
      parsnip::set_mode(mode)

    wf <-
      workflows::workflow() %>%
      workflows::add_recipe(rec) %>%
      workflows::add_model(spec)

    fit <- wf %>% parsnip::fit(data_trn)
    fit_wf <- fit %>% workflows::pull_workflow_fit()

    data_trn_jui <-
      rec %>%
      recipes::prep(training = data_trn) %>%
      recipes::juice()

    x_trn_jui <-  data_trn_jui[, setdiff(names(data_trn_jui), col_y)] %>%
as.matrix()
    y_trn_jui <- data_trn_jui[[col_y]]

    y_trn_jui <-
      if(mode_is_class) {
        as.integer(y_trn_jui) - 1L
      } else {
        y_trn_jui
      }
```

```r
    vip_wrapper_partial <-
      partial(
        vip_wrapper,
        object = fit_wf$fit,
        num_features = x_trn_jui %>% ncol(),
        ... =
      )

    # Returns POS/NEG for glm/glmnet disc
    vi_vip_model <- vip_wrapper_partial(method = 'model')

    # I believe these are the defaults chosen by `{vip}` (although its actual
default is `metric = 'auto'`).
    metric <- ifelse(mode_is_class, 'sse', 'rmse')
    f_predict <- choose_f_predict(engine = engine)

    vip_wrapper_partial_permute <-
      partial(
        vip_wrapper_partial,
        method = 'permute',
        metric = metric,
        pred_wrapper = f_predict,
        ... =
      )

    # # lm method for regression won't work with the general case.
    # vi_vip_permute <-
    #   if(engine == 'glm') {
    #     vip_wrapper_partial_permute(
    #       train = data_trn_jui,
    #       target = col_y
    #     )
    #   } else {
    #     vip_wrapper_partial_permute(
    #       train = x_trn_jui %>% as.data.frame(),
    #       target = y_trn_jui
    #     )
    #   }

    f_coerce_permute <- ifelse(engine != 'glm', function(x) { x },
as.data.frame)
    set.seed(.seed)
    vi_vip_permute <-
      vip_wrapper_partial_permute(
        train = x_trn_jui %>% f_coerce_permute(),
        target = y_trn_jui
      )

    # Note that `vip:::vi_shap.default()` uses `{fastshap}` package.
    set.seed(.seed)
    vip_wrapper_partial_shap <-
      partial(
        vip_wrapper_partial,
        method = 'shap',
        train = x_trn_jui,
        ... =
      )
    vi_vip_shap <-
      if(is_ranger) {
        vip_wrapper_partial_shap(pred_wrapper = f_predict)
      } else {
        vip_wrapper_partial_shap(exact = TRUE)
      }

    # # Removed this part since it's basically redundant with the `{vip}` SHAP
method (which I checked).
    # fastshap_partial <-
    #   partial(
    #     fastshap::explain,
    #     object = fit_wf$fit,
    #     X = x_trn_jui,
    #     ... =
    #   )
    #
    # expl_fastshap <-
    #   if(is_ranger) {
    #     fastshap_partial(pred_wrapper = f_predict)
    #   } else {
    #     fastshap_partial(exact = TRUE)
    #   }
    #
    # # Need to remove the non-diamonds_modified class in order to use {dplyr}
functions.
    # class(expl_fastshap) <- c('tbl_df', 'tbl', 'data.frame')
    #
    # vi_fastshap <-
    #   expl_fastshap %>%
    #   summarize_all(~mean(abs(.))) %>%
    #   # This is actually already `imp_abs`, but it won't matter in the end.
    #   pivot_longer(matches('.'), names_to = 'var', values_to = 'imp')

    # idk why, but I can use `ifelse()` here and return a function that won't
have unexpected output (i.e. a list instead of a dataframe).
    # This is not true for the other `if...else` statements
    f_coerce_dalex <- ifelse(engine == 'xgboost', function(x) { x },
as.data.frame)
    expl_dalex <-
      DALEX::explain(
        fit_wf$fit,
        data = x_trn_jui %>% f_coerce_dalex(),
        y = y_trn_jui,
        verbose = FALSE
      )

    # DALEX::loss_root_mean_square == vip::metric_rmse
    # DALEX::DALEX::loss_sum_of_squares == vip::metric_sse

    f_loss <- if(mode_is_class) {
      DALEX::loss_sum_of_squares
    } else {
      DALEX::loss_root_mean_square
    }
    set.seed(.seed)
    vi_dalex_init <-
      expl_dalex %>%
      DALEX::variable_importance(
        type = 'difference',
        loss_function = f_loss,
        n_sample = NULL
      )
    vi_dalex_init

    # Regarding why `permutation == 0`, see `ingredients:::feature_
importance.default()`, which is called by `ingredients:::feature_
importance.explainer()`, which is called by `DALEX::variable_importance`
    # Specifically, this line: `res <- data.frame(variable = c("_full_model_",
names(res),  "_baseline_"), permutation = 0, dropout_loss = c(res_full, res,
res_baseline), label = label, row.names = NULL)`
    vi_dalex <-
      vi_dalex_init %>%
      as_tibble() %>%
      filter(permutation == 0) %>%
      mutate(
        imp = abs(dropout_loss) / max(abs(dropout_loss))
      ) %>%
      select(var = variable, imp) %>%
      filter(!(var %in% c('_baseline_', '_full_model_'))) %>%
      arrange(desc(imp))

    vi_rnks <-
      list(
        vip_model = vi_vip_model,
        vip_permute = vi_vip_permute,
        vip_shap = vi_vip_shap,
        # fastshap = vi_fastshap,
        dalex = vi_dalex
      ) %>%
      map_dfr(bind_rows, .id = 'src') %>%
      group_by(src) %>%
      mutate(imp_abs = abs(imp)) %>%
      mutate(imp_abs_norm = imp_abs / sum(imp_abs)) %>%
      select(var, imp, imp_abs, imp_abs_norm) %>%
      mutate(rnk = row_number(desc(imp_abs))) %>%
      ungroup()
    vi_rnks
  }

compare_and_rank_vip_q <- quietly(compare_and_rank_vip)
# sysfonts::font_add_google('')
# font_add_google('Roboto Condensed', 'rc')
# sysfonts::font_add_google('IBM Plex Sans', 'ips')

prettify_engine_col <- function(data) {
  res <- data %>% mutate_at(vars(engine), ~sprintf('{%s}', engine))
}

factor_src <- function(x) {
  ordered(x, levels = c('vip_model', 'vip_shap', 'vip_permute', 'dalex'))
}
```

```
plot_rnks <- function(df_rnks, option = 'D') {
  viz <-
    df_rnks %>%
    group_by(var) %>%
    mutate(rnk_mean = rnk %>% mean(na.rm = TRUE)) %>%
    ungroup() %>%
    mutate_at(vars(var), ~forcats::fct_reorder(., -rnk_mean)) %>%
    ungroup() %>%
    prettify_engine_col() %>%
    mutate_at(vars(src), ~ordered(., levels = c('vip_model', 'vip_shap',
'vip_permute', 'dalex'))) %>%
    mutate(lab = sprintf('%2d (%s)', rnk, scales::percent(imp_abs_norm, accuracy
= 1, width = 2, justify = 'right'))) %>%
    ggplot() +
    aes(x = src, y = var) +
    geom_tile(aes(fill = rnk), alpha = 0.5, show.legend = F) +
    geom_text(aes(label = lab)) +
    scale_fill_viridis_c(direction = -1, option = option, na.value = 'white') +
    theme_minimal(base_family = '') +
    facet_wrap(~engine) +
    theme(
      plot.title.position = 'plot',
      panel.grid.major = element_blank(),
      panel.grid.minor = element_blank(),
      plot.title = element_text(face = 'bold'),
      plot.subtitle = ggtext::element_markdown(),
    ) +
    labs(x = NULL, y = NULL)
  viz
}

# .dir_png <- here::here()
# .dir_png <- .dir_proj
export_png <- function(x, dir = here::here(), file = deparse(substitute(x)),
width = 8, height = 8, ...) {
  return()
  path <- file.path(dir, sprintf('%s.png', file))
  res <- ggsave(plot = x, filename = path, width = width, height = height, ...)
}

diamonds_c_rnks <-
  engines_named %>%
  map_dfr(
    ~compare_and_rank_vip_q(
      diamonds_modified %>% select(-grp),
      col_y = 'price',
      engine = .x
    ) %>%
      pluck('result'),
    .id = 'engine'
)
diamonds_c_rnks

lab_title <- 'Variable Importance Ranking'
lab_subtitle_diamonds_c <- 'Continuous Target Variable for Model Prediction of
diamonds Data'
```

```
# require(ggtext)
viz_diamonds_c_rnks <-
  diamonds_c_rnks %>%
  plot_rnks(option = 'A') +
  labs(
    title = lab_title,
    subtitle = lab_subtitle_diamonds_c
  )
viz_diamonds_c_rnks

diamonds_d_rnks <-
  engines_named %>%
  map_dfr(
    ~compare_and_rank_vip_q(
      diamonds_modified %>% select(-price),
      col_y = 'grp',
      engine = .x,
    ) %>%
      pluck('result'),
    .id = 'engine'
  )
diamonds_d_rnks

lab_subtitle_diamonds_d <- lab_subtitle_diamonds_c %>%
str_replace('^.*\\sTarget', 'Discrete Target')
viz_diamonds_d_rnks <-
  diamonds_d_rnks %>%
  plot_rnks(option = 'C') +
  labs(
    title = lab_title,
    subtitle = lab_subtitle_diamonds_d
  )
viz_diamonds_d_rnks

mpg_c_rnks <-
  engines_named %>%
  map_dfr(
    ~compare_and_rank_vip_q(
      mpg_modified %>% select(-grp),
      col_y = 'displ',
      engine = .x
    ) %>%
      pluck('result'),
    .id = 'engine'
)
mpg_c_rnks

lab_subtitle_mpg_c <- lab_subtitle_diamonds_c %>% str_replace('of.*Data', 'of
mpg Data')
viz_mpg_c_rnks <-
  mpg_c_rnks %>%
  plot_rnks(option = 'B') +
  labs(
    title = lab_title,
    subtitle = lab_subtitle_mpg_c
  )
viz_mpg_c_rnks
```

```
mpg_d_rnks <-
  engines_named %>%
  map_dfr(
    ~compare_and_rank_vip_q(
      mpg_modified,
      col_y = 'grp',
      engine = .x,
    ) %>%
      pluck('result'),
    .id = 'engine'
  )
mpg_d_rnks

lab_subtitle_mpg_d <- lab_subtitle_mpg_c %>% str_replace('^.*\\sTarget',
'Discrete Target')
viz_mpg_d_rnks <-
  mpg_d_rnks %>%
  plot_rnks(option = 'D') +
  labs(
    title = lab_title,
    subtitle = lab_subtitle_mpg_d
  )
viz_mpg_d_rnks

cor_by_set_engine <-
  list(
    diamonds_c = diamonds_c_rnks,
    diamonds_d = diamonds_d_rnks,
    mpg_c = mpg_c_rnks,
    mpg_d = mpg_d_rnks
  ) %>%
  map_dfr(bind_rows, .id = 'set') %>%
  group_by(set, engine) %>%
  nest() %>%
  ungroup() %>%
  mutate(
    data =
      map(data, ~widyr::pairwise_cor(.x, item = src, feature = var, value =
rnk))
  ) %>%
  unnest(data) %>%
  rename(cor = correlation)
cor_by_set_engine

cor_by_engine <-
  cor_by_set_engine %>%
  group_by(engine, item1, item2) %>%
  summarize_at(vars(cor), mean) %>%
  ungroup()
cor_by_engine

viz_cor_by_engine <-
  cor_by_engine %>%
  prettify_engine_col() %>%
  mutate_at(vars(item1, item2), factor_src) %>%
  mutate(lab = scales::percent(cor, accuracy = 1, width = 3, justify = 'right'))
%>%
  filter(item1 < item2) %>%
  ggplot() +
  aes(x = item1, y = item2) +
  geom_tile(aes(fill = cor), alpha = 0.5, show.legend = FALSE) +
  geom_text(aes(label = lab)) +
  scale_fill_viridis_c(option = 'E', na.value = 'white') +
  theme_minimal(base_family = '') +
  facet_wrap(~engine) +
  theme(
    plot.title.position = 'plot',
    panel.grid.major = element_blank(),
    panel.grid.minor = element_blank(),
    plot.title = element_text(face = 'bold'),
    plot.subtitle = ggtext::element_markdown(),
  ) +
  labs(
    title = 'Variable Importance Rank Pairwise Correlations',
    subtitle = 'Averaged Over diamonds and mpg Data and Over Continuous and
Discrete Target Variables',
    x = NULL,
    y = NULL
  )
viz_cor_by_engine
```

---

1. After all, I want to make sure my results aren’t sensitive to some kind of bias (unintentional in this case).↩

2. For example, for linear regression models using `lm` from the `{stats}` package, variable importance is based upon the absolute value of the t-statistics for each feature.↩

3. This isn’t an academic paper after all!↩

4. Apologies for using “bland” data sets here. At least they aren’t `mtcars` (nor a flower data set that is not to be named)!↩

5. I’ve made the following modification: (a) I’ve taken a sampled fraction of the data set to increase computation time. (b) I’ve excluded two of the categorical features—`clarity` and `color`, both of which are categorical with a handful of levels. I’ve done this in order to reduce the number of variables involved and, consequently, to speed up computation. (This is just an example after all!) (c) To test how variable importance scores differ for a continuous target variable, I’ll be defining models that predict `price` as a function of all other variables. (d) For discrete predictions, the target is a binary variable `grp` that I’ve added. It is equal to `'1. Good'` when `cut %in% c('Idea', 'Premium')` and `2. Bad'` otherwise. It just so happens that `grp` is relatively evenly balanced between the two levels, so there should not be any bias in the results due to class imbalance.↩

6. Modifications include the following: (a) I’ve excluded `manufacturer`, `model`, `trans`, and `class`. (b) For continuous predictions, I’ll predict `displ` as a function of all other variables. (c) For discrete predictions, I’ve created a binary variable `grp` based on `class`.↩

7. (I haven’t actually checked the source for `{glmnet}` and compared it to that of `lm()`/`glm()`. Differences may arise due to underlying differences in the algorithm for least squares.)↩

8. I should say that I’m using the `{tidymodels}` package to assist with all of this. It really shows off its flexibility here, allowing me to switch between models only having to change-out one line of code!Finally, for variable importance scores (which is really the focus), I’m going to use the following packages and functions.↩

9. Yes, SSE is certainly not the best measure of loss for classification. Nonetheless, when dealing with a binary outcome variable, as is done here, it can arguably be acceptable.↩

10. Don’t be deceived by the fill contours, which are based on the rankings–the number in front of the parentheses.↩

11. I could have split (or “facetted”) in a different way–e.g. by type of target variable instead of by package-function combo—but I think splitting in this way makes the most sense because the type of model—`{glm}`, `{ranger}`, etc.—is likely the biggest source of variation.