Feature Engineering Case Assignment

As usual, I've provided some starter code in the `feature_engineering_case.R` file. You can write all of your code in that file from within RStudio, then save your changes, commit, and push those changes to GitHub before the deadline to submit.

Assignment Instructions

Congratulations on making it past the end-to-end case assignment! Now that you've demonstrated your mastery of the R language, we can be a little more adventurous in our assignments. This assignment will be less prescriptive than some of the previous assignments, but there won't be any data messiness to fight against. Let's jump in!

First, the dataset. This is a publicly available dataset that describes various aspects of different types of wine. We'll use this dataset to practice some feature engineering and model preparation skills. (I know this is BYU and that we don't drink wine here. You're welcome to think of the data as one that describes lemonade. :)) In case it's helpful, many of the things I'll be asking you to do for this assignment have been adapted from a nice (and quite a bit more thorough) analysis that can be viewed here. That analysis was done using python, but there are visualizations and other things done there that might be a useful reference if you're looking for ideas.

Let's start by giving ourselves a nice group variable that we can use with some of the visualizations. Using the `wine_raw` tibble, add a column called `quality_class` that has three values, populated using the following logic: the value should be `1_low` when `quality` is less than 6, `2_med` when `quality` is either 6 or 7, and `3_high` when `quality` is 8 or more. Save the tibble with the added column to a new object called `wine`.

> *After adding the column, you should be able to see the following:*

```
> wine %>% count(quality_class)
# A tibble: 3 × 2
  quality_class     n
  <chr>         <int>
1 1_low          2384
2 2_med          3915
3 3_high          198
```

1. Make sure that you save the tibble (NOT the result of that `count()` code above) to `wine`.

Next let's take care of the missing values in the data. First, let's make a summary tibble (from the `wine` tibble you just made) so we can get a feel for which columns have missing values and how common they are. Create a tibble that shows the following for each column:

- count of non-missing values
- count of distinct values
- count of missing values
- percent of values that are missing (the percent should be displayed in decimal form ... i.e., don't multiply by 100)

Sort the tibble in descending order of the percent missing, and make sure to use the column names you see in the code block below.

> *Hint: This operation can be quite a bit faster if you use a combination of `summarize()` and `across()` along with some pivoting.*

> *Your summary should look like the following in the first three rows:*

```
# A tibble: 15 × 5
   column               non_missing_count distinct_count na_count percent_missing
   <chr>                            <int>          <int>    <int>           <dbl>
 1 fixed_acidity                     6487            107       10         0.00154
 2 p_h                               6488            109        9         0.00139
 3 volatile_acidity                  6489            188        8         0.00123
```

2. Save the tibble as `missingness_summary`.

Now that we know there are 7 columns with missing values, let's impute those missing values with the median for that column. Use whatever logic you'd like to fill in each of the missing values with the corresponding median from that column. Ensure that the missing values have indeed been eliminated, then save the result to `wine_imputed`.

> *Hint: You can also use `across()` inside of `mutate()` to make these changes with a single line of code.*

3. Make sure that you save the tibble (NOT the result of any missingness checks) to `wine_imputed`.

Now let's take care of outliers. Let's first get an overall feel for the shape of these columns using a single, awesome chart. Use `wine_imputed` and your visualization ninja skills to create a plot that *approximately* matches the example plot I've provided in `plots/plot_1.png`. Note that you don't need to worry about colors, exact wording of titles/legends, etc. Just make sure that you see box plots for all 11 numeric features in the dataset (which excludes the `quality` outcome column, by the way) and that your box plots look about the same as mine.

> *Hint: I used the `theme_bw()` theme and a `facet_wrap()` with `ncol = 4`.*

4. Save your plot object to a variable called `plot_1`.

As we can see in the plot, some columns are worse than others in terms of outliers. In other contexts, we might choose which columns to cap vs. which ones to leave, but for the purposes of this assignment, we're going to just cap all of the numeric columns according to an upper threshold. To help us with this, let's make another summary tibble (using `wine_imputed`) that shows both the 95th and 99th percentile values for each of the 11 numeric feature columns. Make sure to use the column names you see in the code block below.

> *Hint: If you choose to just calculate the percentiles for each column manually, you might look into how to use either `tribble()` or define a `tibble()` and then use some piped `add_row()` calls. But it's also possible to do this programatically using a combination of `summarize()` and, optionally, `across()`, along with some pivoting.*

> *Your summary should look like the following in the first three rows. (Note that I sorted mine alphabetically, but that's not required.)*

```
# A tibble: 11 × 3
   column                cap_95   cap_99
```

```
  <chr>                  <dbl>   <dbl>
1 alcohol               12.7    13.4  
2 chlorides              0.102   0.186
3 citric_acid            0.56    0.74 
```

5. Save the summary tibble as `cap_reference` .

Now let's evaluate which cap to apply to the columns. Starting with `wine_imputed` , we're going to make a summary table that shows a count of values that surpass the two cap values we've calculated for each column. Doing this manually would be really dumb, so let's step through an efficient way to do this.

First, pivot the values from those 11 columns so that we can make the `cap_reference` tibble do most of the work for us by joining it in. You'll now have all of the values next to the two relevant cap thresholds, and you can simply `mutate()` two binary columns that flag the values that fall beyond each cap. You don't need to save the data at this point, but I'll give you a preview of what it looks like so you can make sure you're on the right track:

```
# A tibble: 71,467 × 10
  id     type   quality quali…¹ column  value   cap_95  cap_99 is_out95 is_out99
  <chr> <chr>    <dbl> <chr>   <chr>   <dbl>   <dbl>   <dbl>    <dbl>    <dbl>
1 wine… white        6 2_med   alcoh…   8.8    12.7    13.4         0        0
2 wine… white        6 2_med   chlor…   0.045   0.102   0.186       0        0
3 wine… white        6 2_med   citri…   0.36    0.56    0.74        0        0
4 wine… white        6 2_med   densi…   1.00    0.999   1.00        1        1
```

After acheiving the view above, you can simply group and summarize by `column` to get a count of the number of values for each column that will be capped, then mutate a percentage for each (again, decimal percentage, not multiplied by 100). The result should look like the preview below:

```
# A tibble: 11 × 5
  column                 outliers_95 outliers_99 perc_capped_95 perc_capped_99
  <chr>                        <dbl>       <dbl>          <dbl>          <dbl>
1 alcohol                        291          55         0.0448        0.00847
2 chlorides                      317          65         0.0488        0.0100 
3 citric_acid                    311          29         0.0479        0.00446
```

6. Save the summary tibble as `cap_counts` .

The counts we calculated in the previous problem are relatively small and uniform, so we'll conclude that capping the data at either threshold shouldn't be problematic. What would make them problematic, you ask? Great question! Mostly we're making sure that choosing a cap threshold won't affect more than a small number of observations, which can sometimes happen if, for example, there are a bunch of values resting at some maximum value. In that situation, choosing to cap the population at the 99th percentile would affect *more* than the approximately 1% we were assuming. Luckily, the counts and percentages in `cap_counts` don't raise any alarms.

So let's cap each of the 11 numeric feature columns in the `wine_imputed` dataset at their corresponding 99th percentile threshold from the `cap_reference` table. The result should be saved as `wine_corked` (rather than capped...get it? Hilarious.).

> *Hint: You can do this manually (with 11 different mutate statements and hardcoded thresholds), but you can also use some of tools we've been using in the other problems to do the capping much more efficiently.*

7. Make sure that you save the tibble (which will still have the same row and column counts as the previous wine tibbles) to `wine_corked`.

To examine the effect that our capping has had on the distributions, copy the code you used to create the first plot and make a new one, this time based on the `wine_corked` dataset. There should be a noticeable difference between the two plots. Again, your plot should *approximately* match the example plot I've provided in `plots/plot_2.png`.

8. Save this new plot object to a variable called `plot_2`.

Next up: testing for skewness. You'll see that I've left a handy function in the case script. This function will calculate a measure of skewness for you, and you can use it just like any other column function (like `mean()` or `min()`). Use that function to create a summary table that summarizes the skewness for all of the numeric feature columns in `wine_corked`. Arrange the tibble in descending order of skewness.

> *Your summary should look like the following in the first three rows:*

```
# A tibble: 11 × 2
  column                skewness
  <chr>                    <dbl>
1 chlorides                 2.21
2 fixed_acidity             1.40
3 volatile_acidity          1.27
```

9. Save the summary tibble as `skewness_summary`.

Different people have different definitions of what "too much skewness" is, but we're going to consider any above 0.7 to be high and deserving of a log transformation. You'll see in the `skewness_summary` that there are 5 such columns. There are many ways to take care of non-normal data, most of which we will explore when we work through the modeling pipeline in a week or two. For now, we'll just stick with a basic log transformation (accomplished with `log()`). But before you do that, notice that 3 of the 5 skewed columns have values between 0 and 1. These won't really be affected by our logarithmic transformation (since the logarithmic math changes small numbers very little). In the future, we'll evaluate other power-based functions, but for now we're just going to help our transformation be more effective by first multiplying those three columns by 100. That way, the log function will have a more convincing reshaping effect and we'll be able to move on. (And don't worry about us hacking these numbers around a bit...we're eventually going to standardize their scales anyway; this is sometimes how the sausage gets made.)

Starting with `wine_corked`, go ahead and do the transformations as described above. (In the next step we'll again recreate the box plots so you can check your work.)

10. Save the tibble above to `wine_transformed`.

Just as before, copy the code you used to create the first two plots and make a new one, this time based on the `wine_transformed` dataset. There should be a noticeable difference between plots 2 and 3 (though obviously only for the 5 columns that we just transformed). Again, your plot should *approximately* match the example plot I've provided in `plots/plot_3.png`.

11. Save this new plot object to a variable called `plot_3`.

If you look carefully at `plot_3`, you'll notice that, while we've made some great progress in pulling in outliers and bringing the distributions closer to normal/even, the scales are quite different from feature to feature. (Notice, for example, how small the range is for the `density` feature.) As we discussed in class, we'll make our model happier if the scales for the variables are closer to the same.

Luckily, the process for doing so is a single function: `scale()`, which can be applied to any column with no other parameters to convert that column to a z-score. Use that function on the `wine_transformed` data to standardize all of the numeric *features* (not the `quality` column!).

12. Save the tibble to `wine_scaled`.

Again, copy the code you used to create the other plots and make a new one, this time based on the `wine_scaled` dataset. You should now see nice, evenly scaled features that are centered around 0. Again, your plot should *approximately* match the example plot I've provided in `plots/plot_4.png`.

13. Save this new plot object to a variable called `plot_4`.

The final step in our journey will be to check for multicollinearity among the numeric features. We'll do this in two steps. The first will be to use the `ggcorrplot` package to generate a correlation plot that summarizes all of the correlations among the numeric variables in the `wine_scaled` tibble. Teach yourself how to use that new package to generate a plot that matches the example plot I've provided in `plots/plot_5.png`.

> *Just to be clear, I haven't used any additional options other than the obvious ones that you'll use to get your plot to look like mine. The only nuance I'll note is that `ggcorrplot`, by default, orders the columns in the plot according to how they come in from the data, so you may need to do some reordering beforehand.*

14. Save this new plot object to a variable called `plot_5`.

We aren't actually going to do any removing of features because we're not going to proceed to model with this data (yet!). But just take note of the features that are highest in terms of correlations. (Both positive and negative correlations are relevant.) We'll revisit these correlations later when we use this data to practice some modeling.

For now, we'll do one last check for multicollinearity: the Variance Inflation Factor (VIF). To do this, you'll need to do the following:

1. build a simple linear model (using the `lm()` function) with the `quality` variable regressed on all 11 numeric features in `wine_scaled`.
2. the result of that model is then passed (with or without saving first, either way is fine) to the `vif()` function from the `car` library.

The result of the `vif()` function is a numeric vector (with names) that shows the VIF for each column. Again, we're not going to act on this information now, but take note of the features that have high (meaning more than about 5 or 10) VIF values. Those will be our first candidates for removal when it comes time to test different modeling scenarios.

Here are a few of the VIF values you should see in your result:

```
  alcohol           chlorides
 4.742267            2.161442
```

15. Save that named vector to `vif_results` . And you're done!

Final Cleanup and Submission

Final Run

Please do the following to make sure your code runs without errors and passes the naming checks by doing the following:

1. Restart your R session (Session >> Restart R).
2. Run your entire script from the beginning, watching for any errors along the way.
3. After your script is finished, make sure that you pass all tests in the code in the "Naming Checks" section at the bottom of your script.

Housekeeping

Once you have completed the steps in the prior section, please check all of the following and make adjustments as needed. (Failure to do the housekeeping steps below will likely cause an error in our grading process, and that will make it hard to give you the right credit for your hard work.)

1. If you used the `setwd()` command near the beginning of the script, please COMMENT OUT that line before committing and pushing your code.
2. Please also comment out any code where you are either using the `View()` or `glimpse()` functions. These cause issues with our grading procedures.
3. If you installed any new packages as you completed the assignment (using `install.packages()`), please comment those installation commands out as well.
4. Lastly, please ensure that you have **NO** absolute references in your code that are not commented out. (An absolute reference is something like: `/Users/YourName/Documents/GitHub/...` or `C:/GitHubProjects/...` .) These also throw errors and make it hard for us to grade your work.

Save, Commit, Push

You're now ready to do all three of the following:

1. Save your script.
2. Do a final commit with your git tool to stage your work for submission.
3. Push your changes up to your repository. And you're done!