Modeling Pipeline Case Assignment

As usual, I've provided some starter code in the `model_pipeline_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

Well, my friends. You have arrived at your final (individual) challenge. Complete this last case assignment, and you will be granted the rank of Full Data Science Ninja. This assignment will take you through 3 separate model-building pipelines of increasing complexity, culminating in a real-world model-building scenario that you will take through the full hyperparameter tuning process. Excited? Me too. Let's do it.

First, though, a quick note about setting seeds. There are several places where you'll need to use the same seed that the grader uses in order for your results to line up properly. To avoid any confusion or uncertainty, I have included each seed command where it's required, along with comments to pinpoint the process that each seed is intended to standardize. You'll also notice that I'm using the exact same seed (42, of course), each time it's set. So just take care not to add any *extra* random processes, and everything should line up nicely. (That said, the third modeling excercise won't be graded as tightly, so you probably don't need to worry too much about seeds there.)

The first model you'll be building will use the wine dataset from our last case assignment. You'll be predicting `quality`. Start by removing the `id` column from `wine_raw`, saving the result to `wine`. Then use that `wine` tibble to create a `wine_split` object, followed by the corresponding `wine_training` and `wine_testing` tibbles. (Just make sure your code for this section comes *after* the `set.seed(42)` I provided.)

1. After setting up your model pipeline, you should now have `wine`, `wine_split`, `wine_training`, and `wine_testing` objects in your environment.

Next, you'll be creating a recipe to handle most of the feature engineering work that we went through in the last case assignment. Create a recipe (called `wine_rec`) with steps that handle the following:

- median imputation for all numeric features
- Yeo Johnson transformations for all numeric features
- standardization (z-score conversion) for all numeric features
- dummy coding for the `type` column

Once you have created your recipe as described above, train the recipe with `wine_training`, saving the result to `wine_rec_prep`. Then apply the processing steps to `wine_training` and `wine_testing`, saving the results to `wine_training_baked` and `wine_testing_baked`, respectively.

2. After accomplishing the above, you should now have `wine_rec`, `wine_rec_prep`, `wine_training_baked`, and `wine_testing_baked` objects in your environment.

Next, create a model specification for a basic linear regression model using the standard "lm" engine. Save the specification as `wine_lm`, then fit the training data with that model, saving the result as `wine_lm_fit`.

3. After accomplishing the above, you should now have `wine_lm` and `wine_lm_fit` objects in your environment.

Now generate predictions for the testing data (saved to `wine_pred`), and combine those predictions with the (original, not baked) test data (saved to `wine_testing_results`). Note that the data used for both the `fit()` as well as the `predict()` functions should be the "baked" datasets you just finished baking. Use the standard functions from the `tidymodels` package to calculate the RMSE and the R-squared model performance metrics. Ensure that both of these metrics are stored together in a single, 2x3 tibble called `wine_performance`. Then use the data from `wine_testing_results` to generate a plot (named `plot_1`) that looks similar the plot I've saved as `plots/plot_1.png`.

4. After accomplishing the above, you should now have `wine_pred`, `wine_testing_results`, `wine_performance`, and `plot_1` objects in your environment.

On to the next dataset! This one summarizes passengers' reported satisfaction with their experience during a flight. The target variable is `satisfied`, a binary indicator of whether they were satisfied or not. The other columns are all features related to the flight and/or the passenger.

Start by setting up the model and creating the train/test split and deriving both the training and testing tibbles. You can name each of those `air_split`, `air_training`, and `air_testing`, respectively.

5. After accomplishing the above, you should now have `air_split`, `air_training`, and `air_testing` objects in your environment.

We're going to look at one method for comparing different data preprocessing setups using different recipe configurations. Specifically, you'll create two different recipes that are identical except for the last step. Both recipes will have the following:

- median imputation for all numeric features
- Yeo Johnson transformations for all numeric features
- standardization (z-score conversion) for all numeric features
- dummy coding for the category features
- a near zero variance filter to remove features that have very little variance

Again, both recipes should have the steps above. Then, as the last step on each recipe, you'll add two different approaches for dimensions reduction. The first recipe (saved as `air_rec_corr`) will use a correlation-based filter (with default threshold) while the second recipe (saved as `air_rec_pca`) will use a principal components analysis to reduce the many columns to several principal components.

In the next steps, we're going to use some workflow objects to pair these two different recipes with two algorithms to explore which dimension reduction approach provides better performance. Before we do that, however, I would encourage you to `prep()` and `bake()` each one with at least the training data, just to examine the result of those two dimension reduction strategies. There is no need to save the result of those operations (and the grader will ignore them if you do), but it would be good for you to have an understanding of what a PCA actually does. You'll notice that it produces a *very* different end result.

6. After accomplishing the above, you should now have `air_rec_corr` and `air_rec_pca` objects in your environment.

Next, let's set ourselves up to compare two different modeling algorithms. Create a model specification for a logistic regression (default engine) and save it as `lr_model`. Next, create a model specification for a boosted tree ('xgboost' engine) and save it as `xgb_model`.

7. After accomplishing the above, you should now have `lr_model` and `xgb_model` objects in your environment.

Rather than doing the data processing and model fitting process manually, let's instead create some workflows to do most of the work for us. We're going to create a total of 4 workflows, pairing each recipe with each model specification. These workflows should be named `lr_corr_wkfl` , `lr_pca_wkfl` , `xgb_corr_wkfl` , and `xgb_pca_wkfl` .

8. After accomplishing the above, you should now have `lr_corr_wkfl` , `lr_pca_wkfl` , `xgb_corr_wkfl` , and `xgb_pca_wkfl` objects in your environment.

Now let's do an overall evaluation of each of the four workflows in terms of how they compare on the standard classification performance metrics calculated for the test data. Use the workflows you defined in the prior step to fit and generate test-set performance metrics. (Note that you should use the seed provided before running the xgboost models since their execution uses a randomized learning process.) Save the resulting fit objects as `lr_corr_fit` , `lr_pca_fit` , `xgb_corr_fit` , and `xgb_pca_fit` . Then construct a (single) result comparison tibble that contains measures of accuracy and roc_auc for each of the 4 workflows, using the labels you see in the preview below. Arrange the tibble by metric and (descending) by estimate and save the tibble as `air_models_compared` .

> *Hint: Remember that there is a convenient way to extract performance metrics from a workflow that has been fit. Combining that extraction function with a `bind_rows()` will be a good way to get this done.*

```
# A tibble: 8 × 5
  .metric  .estimator .estimate .config              model
  <chr>    <chr>          <dbl> <chr>                <chr>
1 accuracy binary         0.952 Preprocessor1_Model1 xgb_corr
2 accuracy binary         0.881 Preprocessor1_Model1 xgb_pca
3 accuracy binary         0.879 Preprocessor1_Model1 lr_corr
4 accuracy binary         0.825 Preprocessor1_Model1 lr_pca
```

Which algorithm appears to perform better? Which recipe approach? These are thought questions for you (and you don't need to provide answers in any way in your environment).

9. After accomplishing the above, you should now have `lr_corr_fit` , `lr_pca_fit` , `xgb_corr_fit` , `xgb_pca_fit` , and `air_models_compared` objects in your environment.

Now let's do a more robust cross validation test using the workflow that performed the best in terms of **roc_auc** in the `air_models_compared` tibble. Setup a 10-fold cross-validation (saved to `air_folds`), then use the best-performing workflow to obtain cross-validated modeling results, saving the results to `cv_fit` . When you set up your cross validation, use 3 different repeats, each with 10 total folds.

Next, construct a summary tibble that displays the minimum, average, and maximum estimates for both of the two default performance metrics. This will give you a slightly more complete picture of the consistency of performance across the 10 folds within each repeat. Save the resulting tibble to `cv_perf_summary` . (A preview of the top row of `cv_perf_summary` is shown below.)

> *Hint: there is a parameter option in `collect_metrics()` that will allow you to extract detailed performance metrics for each of the many models you ran rather than the summarized set of metrics.*

```
# A tibble: 2 × 4
  .metric  minimum  mean maximum
  <chr>       <dbl> <dbl>    <dbl>
1 accuracy    0.944 0.950    0.957
```

How does the performance pattern you're seeing across the various folds compare to that obtained in the `air_models_compared` tibble from the last exercise? Would you say that the performance is consistent or is there cause for concern? (Again, these are thought questions that you can consider but no need to save an answer in the environment.)

10. After accomplishing the above, you should now have `air_folds`, `cv_fit`, and `cv_perf_summary` objects in your environment.

Okay. Final dataset and model. This data comes from the lending activities of Lending Club (a fintech lender I worked with at Neuro-ID), and each row in the dataset represents a loan that was issued during the year 2019. The target variable is `loan_default`, and your job is to build and then tune a model that helps predict whether a loan is going to be repaid or not.

On purpose, I'm not going to be very prescriptive on this last exercise. You should be less concerned about whether you've set up or tuned your model exactly the same way as I have. I'm more interested in you thinking through the modeling pipeline process and using what we've learned in class to get to a model that is ready to deploy.

Below, I'll list each of the objects that I ended up with after building and tuning this loan default model. First, though, here are a few things to keep in mind:

- We're going to be running a lot of models during the tuning process, so to speed things up we are going to use xgboost and very little pre-processing in terms of recipe steps. (Xgboost is actually very robust in terms of data shape, missingness, etc., so we don't have to worry as much about getting the data perfect for the model; either way, removing a bunch of processing will save you some time during the grid search, so that's what we'll be doing.) So when you build your recipe, I would suggest that you add just one pre-processing step: `step_dummy(all_nominal_predictors())`.
- I'm setting my seed each time I (a) do a random assignment (e.g., `initial_split()` or `vfold_cv()`), (b) run any xgboost fit procedure, or (c) when I generate my tuning grid. (Again, I'm not going to be too worried about whether your model results are the same, so it won't matter as much for this exercise.)
- It's usually a good idea to do an initial, cross-validated model with the "default" xgboost parameters, and then add a "tunable" model spec that you use for tuning and comparison. You'll see in the list below that I've taken this approach.
- The full tuning process can take a while, and you don't want to be waiting around for your code to run (especially if you're working on this at the last minute). I would suggest keeping your search grid to 10 or fewer (even though that's smaller than what we would usually do out in the wild).
- You can also speed the tuning up considerably if you do the grid search in parallel on multiple processor cores. There is a wonderfully usable way to do this in the form of a package: `doParallel`. After installing the package, you can initiate a parallel process by calling `doParallel::registerDoParallel()` before you do your `tune_grid()` operation. This works a bit more smoothly on Mac/Linux machines than on Windows machines, but either way I'd recommend playing around with it a bit (and monitoring your processor consumption while it's running) because it's somehow just *fulfilling* to see your machine firing on all cylinders. (And if you don't want to mess with this, you don't need to; it'll just mean more waiting while your computer runs through the grid search sequentially. Obviously you can reach out on Slack if you have questions about this.)

- As we discussed in class, the xgboost algorithm has a lot of tunable settings. If you want some background reading on what the various settings do, you can read about them here. Just try not to read through the parameter descritpions shortly before going to bed, because they are **so** exciting that you'll probably get all worked up and then not be able to sleep.
- Within the `tidymodels` implementation of the xgboost algorithm, you can see the main parameters available for tuning in the documentation for the `boost_tree()` function. I would avoid tuning the `mtry` or `trees` parameters. Tuning the `trees` parameter in particular will increase your processing time *significantly*, so if you're short on time you should **definitely** leave that one out. (More trees will likely get you better performance, but I wouldn't use more than about 300-500 trees regardless.)

Okay. With the above in mind, here's a list of the objects that I used to build out my model pipeline, and I'm listing them in the order that I added them to help guide you a bit:

- `lc_split` : the initial 75/25 split object
- `lc_training` : the training data
- `lc_testing` : the testing data
- `lc_rec` : the (single-step) recipe
- `xgb_model_default` : xgboost model spec with default parameters
- `lc_xgb_wkfl` : workflow comprised of `lc_rec` and `xgb_model_default`
- `lc_folds` : the 10-fold cross-validation object
- `lc_default_fit` : the fit of the 10-fold cross-validation using the default workflow
- `xgb_model_tuning` : a different xgboost model spec with tunable parameters
- `lc_grid` : the tuning grid (again, I would keep this sized to 10 or so unless you have lots of breathing room). You can use `grid_random()` to generate this, but there are others out there as well.
- `lc_xgb_tune_wkfl` : a different workflow that bundles `lc_rec` and `xgb_model_tuning`
- `tune_fit` : The result of the grid search (i.e., `tune_grid()`)
- `best_parameters` : the best-performing parameter set from your grid search
- `final_lc_wkfl` : a final modeling workflow, derived from applying those `best_parameters` to the `lc_xgb_tune_wkfl` workflow.
- `lc_final_fit` : The end result, a deployable model object that results from finalizing the `final_lc_wkfl` with the `lc_split` object.

> *Bonus Reward: If you are having fun with this and want to see how high you can get your model's performance, then go for it. I will gladly reward the top 2 students whose model performance is the highest across all three sections. The reward is your choice of either (a) extra credit, (b) I'll buy you lunch, or (c) I'll fund date night for you and your +1 in the form of a restaurant gift card. Performance will be measured in terms of roc_auc, and I'll maintain a "leaderboard" of the top 5 AUCs that have been acheived and share those on Slack periodically until the assignment deadline. (To report your auc, just slack me with a screenshot of your console output after running* `tune_fit %>% show_best()` *).*

11. After accomplishing the above, you should now have *all of the objects listed above* in your environment. (I'm not going to list them here...)

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!