Milestone 2: Group Model Building

With your group's data selected and your business case made, your next task will be to train your first model and get it ready for deployment. This deliverable will ask you to collaborate as a group to build a single script that takes your group's assigned data through a full model training exercise, then saves the result out for deployment.

Dataset

As a reminder, your group has been assigned a specific subset of data to use for this milestone. You'll download this source data using the link provided on this page on Learning Suite. Importantly, you will not be using the data available on Kaggle. You should be able to just read the file straight from that URL using `read_csv()` like we do for other assignments.

(The data you'll be using is a random subset of the total labeled data available from Kaggle, and I'm purposefully holding out a test set so that we can go through the process of evaluating "new" data for the next milestone.)

Milestone Requirements

Your task for this milestone will be quite similar to the individual modeling pipeline assignment, and you'll be expected to thoroughly build, iterate, evaluate, and tune your model as if your job depended on the quality of the model. :)

Data Familiarity / Exploration / Cleaning

After reading in the data, you'll want to do some due diligence with the data just to make sure that you have a good feel for what columns are available to you, their general behavior, etc. While data exploration and cleaning are not the focus of this milestone, your final project milestone and presentation will require you to demonstrate mastery over your data, and you'll be expected to answer relatively piercing questions about the dataset you chose. So spend some time with the data, build some visualizations, run some correlation analyses, and otherwise just get a feel for what is in your data. You don't need to save or deliver anything from this exploration for this milestone, but your analysis should inform some of what you do in terms of feature engineering, algorithm selection, etc.

(Note that because we've pulled all of the datasets from Kaggle, you are welcome to make use of other users' exploratory data analyses that you can find.)

Train/Test Split

You'll need to start your modeling pipeline as usual with an initial split, then derive the training and testing tibbles from that split object.

1. Please name the split object, training set, and testing set `data_split`, `data_training`, and `data_testing`, respectively.

Feature Engineering Recipes

You should use at least 2 different recipe approaches to try out some different feature engineering and/or feature selection procedures. These recipes can follow conventions we've used in class, or you can explore new ones and see which combination of engineering procedures perform best with your data. You should

have a reasoned argument prepared about why you designed your recipes the way you did. (I'll ask you to explain these a bit as a part of your final deliverables later in the semester.)

2. Please name each of your recipes using the following convention: `recipe_1`, `recipe_2`, etc.

Model Algorithm Specifications

You should also use at least 2 different classes of algorithm (e.g., logistic regression vs. random forest vs. decision/boosted tree). For one of these classes, you should try out two different computation engines (e.g., you could test out both the "ranger" and "randomForest" engines for the random forest algorithm). This means that you'll need to use a **minimum of 3 different model specifications** as a part of your model exploration process.

3. Please name each of your model specifications using the following convention: `model_spec_1`, `model_spec_2`, `model_spec_3`, etc.

Workflow Objects

You should create workflows that combine each recipe with each model specification so that you can test your different feature engineering approaches with your different modeling algorithms/engines. This means that you'll have **a minimum of 6 workflows**. (Don't worry...we'll be reducing our focus to just one of these in just a few steps.)

4. Please name each of your workflows using the following convention: `r1_m1_wkfl`, `r1_m2_wkfl`, `r1_m3_wkfl`, `r2_m1_wkfl`, `r2_m2_wkfl`, etc.

Initial Cross-Validated Results

You'll then take each of your workflows through a cross-validated fit process. For this, you'll need to create an object using `vfold_cv()` (named `data_folds`) and then fit each workflow using cross-validation.

Before testing all of the workflows using cross-validation, use the `metric_set()` function to define a custom set of performance evaluation metrics that you as a group feel would be the most important metrics to measure and compare. (We haven't used the `metric_set()` function yet, but you can read about how to use it here). Note that you'll need to select performance metrics that align with the type of model you're building, and you should choose at least three. (Be prepared to justify why you selected the performance metrics you chose.)

> *Note: If your model is a classification model, be sure to include the `roc_auc` metric in your custom set. This will ensure that you can build the required plot at the end.*

Then, using the same folds object for all workflows, you should fit each of your workflows to your cross-validation dataset, saving each process to its own result object. (Note that the `fit_resamples()` function has a parameter called `metrics` that will allow you to pass your set of custom metrics to the CV process. Doing so will allow you to extract those custom metrics after the fitting process is complete.)

Lastly, use the `collect_metrics()` function to gather your customer performance metrics from each of those cross-validated modeling runs, assembling a single comparison tibble (named `workflow_perf_summary`)that summarizes the performance of each workflow. Ensure that this tibble has, in addition to the standard set of columns returned by `collect_metrics()`, three additional columns: `workflow_name`, `algorithm_name`, and `engine_name`. The `workflow_name` column should correspond to the actual object name of the workflow from the previous step (e.g., `r1_m1_wkfl`), and the

`algorithm_name` and `engine_name` columns should provide the name of the algorithm and engine tested in that workflow, respectively.

After assembly, the `workflow_perf_summary` tibble should look something like this:

```
# A tibble: 24 × 9
   .metric  .estimator  mean     n std_err .config              workflow_name
algorithm_name engine_name
   <chr>    <chr>      <dbl> <int>   <dbl> <chr>                <chr>         <chr>
<chr>
 1 accuracy binary     0.741    10 0.00712 Preprocessor1_Model1 r1_m1_wkfl
logistic_reg   glm
```

5. Please name the folds object `data_folds`, the custom metrics object `custom_metrics`, the performance summary tibble `workflow_perf_summary`, and each of the fit result objects using the following convention: `r1_m1_fit`, `r1_m2_fit`, `r1_m3_fit`, `r2_m1_fit`, `r2_m2_fit`, etc.

Promising Algorithm (Workflow) Selection

Using the metric or metrics that would be more important for your business' use case, select the most promising workflow from the `workflow_perf_summary` tibble as the one you'd like to optimize and finalize with. Ensure that the one you choose contains an algorithm that has "tunable" parameters so that we can take it through the tuning and optimization process. Make a copy of the fit object selected and name the "to-be-tuned" workflow fit as `best_initial_workflow_fit`.

6. In other words, you'll create a new "fit" object that is a simple copy of whatever workflow fit object you selected from the comparison table, using a simple assignment statement like this: `best_initial_workflow_fit <- r2_m1_fit`.

Hyperparameter Tuning

Now you can create a new workflow that uses the same algorithm, engine, and recipe as that found in `best_initial_workflow_fit`, except this time you'll include tuning parameters in your model specification. Save this new "tunable" model specification as `model_spec_tunable`, and the associated tunable workflow as `tunable_wkfl`. (Optionally, you could explore tuning some of the parameters in your chosen recipe, saving the new recipe as `recipe_tunable` before including it in the `tunable_wkfl` workflow, but that's not required.)

Now create a tuning grid of at least 10-20 rows, derived from the tunable parameters you setup in the `tunable_wkfl`. Save this grid as `tuning_grid`.

Finally, use your `data_folds` object from before to do a hyperparameter tuning operation with your `tunable_wkfl` and your `tuning_grid`. Importantly, you should also use your `custom_metrics` from your earlier cross-validation analysis (which can similarly be passed to the `tune_grid()` function using the `metrics` parameter). This will ensure that your custom metrics are calculated and can be used to select your optimal model after the tuning process. Save the resulting fit object as `tunable_fit`. (Try not to melt anyone's computer while doing so.)

7. Please name the new model specification `model_spec_tunable`, the new workflow `tunable_wkfl`, the tuning grid `tuning_grid`, and the result of your tuning process `tunable_fit`.

Finalize your Model

Following your tuning process above, you'll have all of your results summarized in `tunable_fit`. From this object, you can do the following:

- Using one of the metrics you asked for in your `custom_metrics` set (ideally the one you think is most relevant to your business case), have the `show_best()` function show you the top-performing tuning parameters. (Note that `show_best()` has a `metric` parameter that accepts a string containing the name of the metric you'd like to sort by in order to determine what the "best" model is.)
- Similarly, use the same `metric` with the `select_best()` function, saving the result as `best_parameters`.
- Finalize your `tunable_wkfl` with these `best_parameters`, saving the finalized workflow as `finalized_wkfl`.
- Then do your final `last_fit()` operation to train your final, optimized workflow using your original `data_split` object. Save the result to `final_fit`.

8. After finalizing, you should have added `best_parameters`, `finalized_wkfl`, and `final_fit` to your environment.

Summarize Model Performance

It's a good idea at this point to see whether your tuning made much of an improvement. You can quickly do this by comparing the output of `collect_metrics()` on the `best_initial_workflow_fit` object vs the `final_fit` object you just finished creating. (Note that even if there was little or no improvement, that's okay; you won't be graded on anything related to model performance.)

Either way, you'll need to create a single chart that summarizes the performance of your model for the test set. (It's simple to extract the required information from your `final_fit` object using `collect_predictions()`.) If your model is a classification model, that plot will be a standard ROC curve plot. (See the tips on using `autoplot()` after the `roc_curve()` function here if you need a reminder on how to make one of those.) If your model is a regression model, you'll instead build a scatter plot that shows your predicted values vs. your actual values. (You can look at the suggestions here if you need some help doing this with the automatic results functions, see the brief discussion here.)

Your plot won't really be graded on anything other than whether it has the correct plot *type*, but it would be a good idea to add at least a title to your plot so you can easily differentiate from this and one future plot that I'll have you create for the next milestone. Save your plot to an object called `perf_plot`, then use the following code to save that plot to your working directory as a `.png` file:

```
perf_plot %>% ggsave('perf_plot.png', plot = .,
                     device = 'png', width = 14, height = 9)
```

In addition, you'll want to save your test set predictions out as a separate dataset so we can use them later. So use `collect_predictions()` to extract the tibble of test results, saving them as a tibble named `test_results`. Then use `write_csv()` to write that tibble out to disk with the filename `test_set_results.csv`.

> *Note: You'll want to make sure that these two files (as well as the ones that you'll save in the section that follows) are saved to your group's repository folder for this assignment. You'll likely need to use `setwd()` here before you save the files to make sure that happens.*

9. After completing this section, make sure that you now have the `perf_plot` and `test_results` objects in memory, and ensure that you also have `perf_plot.png` and `test_set_results.csv` saved in your group's repository folder for this assignment.

Prep for Deployment

We're going to go through the final few steps that you would follow if you were delivering this model to an engineering team for deployment in an API. Although we won't complete the deployment, your group is going to do the handful of preparation activities that the data science team would typically be responsible for. Don't worry: I'll walk you through it.

You should already have installed the following two packages (if necessary) because they are included at the top of your group's starter script: `tidypredict` and `yaml`. Make sure you've "librariied" them so you have the functions that you need for this section.

Using your `final_fit` object, extract the trained workflow (using `extract_workflow()`) and the fit model object (using `extract_fit_parsnip()`), storing the extractions in `trained_workflow` and `model_object`, respectively. Then use the `parse_model()` function to parse the `model_object` you just extracted, saving the result as `parsed_model`.

Now you can use the command below to write your parsed model out to disk as a `yaml` file. This file is typically the primary "handoff" deliverable, which the software engineers could then build into an API endpoint.

```
write_yaml(parsed_model, "model_config.yml")
```

I'd encourage you to open up the `yaml` file after you write it out so you can see what it looks like. Depending on the algorithm you ended up with, it'll look slightly different for each group, but you should be able to understand how an API could parse the configuration from that file and do some simple math to generate predictions (independent of R or the `tidymodels` framework, etc.). If you want to see what some of the math might look like, you can also run the command below, which will print out a `dplyr`-like `mutate()` command that will give you an idea of some of the math and conditional logic that would be used to actually do the calculation. If your model is relatively complex, the statement will be massive, but you can get a sense for what's happening by scrolling through it a bit.

```
tidypredict_fit(parsed_model)
```

Lastly, you'll want to ensure that you save out the relevant things that you ended up with in your R session so that they can be loaded and used in the future (within a future R session) by your data science team without having to re-run your entire model training procedure. For our purposes, this includes the following things, which should now be available in your environment: `final_fit`, `trained_workflow`, and `model_object`, which you should save to disk as `final_fit.rds`, `trained_workflow.rds`, and `model_object.rds`, respectively. You can use the handy `write_rds()` function to accomplish this, and you might want to use `read_rds()` to read back in one of the `.rds` files you just wrote out, just to make sure that it worked as expected.

> ***Important**: Your next group milestone will require you to read those objects back into memory and use them for a few operations, so you want to be extra sure that they are saved and not lost. If you do lose them, it's not the end of the world because you'll always be able to re-run your model training script.*

> *But you can save yourself a lot of processing time by simply building your next script to read those objects in from disk by reading the `.rds` files using `read_rds()`.*

10. After completing this section, make sure that you now have the `trained_workflow`, `model_object`, and `parsed_model` objects in memory, and ensure that you also have `model_config.yml`, `final_fit.rds`, `trained_workflow.rds`, and `model_object.rds` saved in your group's repository folder for this assignment.

Read this before submission

Your group will submit this assignment in the same way that you've submitted other coding assignments: by committing and pushing your changes. Note, however, that `.rds` files are much larger than a typical script or other similar file. Github does *not* play well with large files, so you'll notice that I've excluded `.rds` files using the `.gitignore` in this repository. I'm pointing this out to you so that you are aware that those files **will not be saved unless you manually archive them somewhere**. Your next group milestone will take significantly longer if you have to redo all of your model training because the `.rds` files went missing. Again, wouldn't be the end of the world, but it'll save everyone a lot of stress and hassle if you just make sure to backup your `.rds` files somewhere that is mutually accessible for your group. (A shared Box folder would be a fine location, but I'll leave it up to you.)

Usually, we ask you to restart your R session and re-run your entire script from the beginning to ensure that you don't run into any errors, but since re-running your script from an empty session will take a LONG time because of all the model training and tuning, I'll instead revise my advice to the following:

1. Ensure that your code passes all tests in the code in the "Naming Checks" section at the bottom of your script. (I re-wrote the tests to also include a check for the required files on disk in your working directory.)
2. Recruit 1 or 2 of your group members who weren't as involved in actually typing up the code toward end to carefully go through your group's code and manually check that everything looks right and that you have written everything out as intended.

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!