Lab 12 – hyperparameter tuning

Submission: `lab12.ipynb` pushed to your Git repo on `master`.
Points: **15**
Due: Friday, April 10
Pair programming: Optional, but strongly recommended

Objectives

- Understand the impact of hyperparameters with model induction and performance
- Learn to use the `GridSearchCV` class to find good parameters for your model

Requirements

- Submit your work as an `.ipynb` file script named **`lab12.ipynb`**.

Background

You were introduced to Keras deep learning framework in the last lab. You also learned how to use the Keras wrapper class, `KerasClassifier`, so that you could continue to use the wonderful **scikit-learn** framework for data mining and machine learning, leveraging Keras within the environment you know. You also learned that there are an enormous number of parameters to consider to get a good result! Yeah, you could do it manually by creating multiple loops to evaluate different model parameters, or you could use the scikit-learn's framework to do this.

Some of this lab is modeled after some code given in a few different web sites, and you might get more out of the lab by taking a few minutes to page through these pages. You might even want to just keep these pages open in your browser.

- First, the most important page you'll need is the reference to `GridSearchCV`: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html
- The best example code can be found right on the scikit-learn site: https://scikit-learn.org/stable/auto_examples/model_selection/plot_grid_search_digits.html
- The following page illustrates a bit simpler code to get you started: https://machinelearningmastery.com/use-keras-deep-learning-models-scikit-learn-python/.

That's it!

The classic Wine dataset

There have been many datasets appearing over the past 15 years or so that attempt to try and capture numerous objective measures to characterize wine. An often-used dataset for classification exercises is the infamous wine dataset from the early 1990s. Not quite as popular as iris, but pretty close. It's a bit more complex than iris, with 13 variables and 1 target multi-class variable.

First, open the following page: https://archive.ics.uci.edu/ml/datasets/wine This page will give you the data, and the information you need to start. (If you prefer, it is also one of the built-in datasets available in `sklearn.datasets` package (see https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_wine.html). Either way, you need to first convert this to a pandas data frame.

The following code will read in the data from the UCI machine learning site, as a data frame in its raw form:

```
names = ["target","alcohol","malic_acid","ash","alcalinity","Mg","tot_phenols","flavanoids",
         "non_flav_phenols","proanthocyanins","color_intensity",
         "hue","OD","proline"]

df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-
databases/wine/wine.data',header=None,names=names)
```

This is a lab that is a bit more hands off than the previous lab was. You will need to do some essential EDA to understand the distributions of your variables with respect to each target class. However, there are surely many examples out there already that demonstrate reading and modeling these exact data.

NOTE: As usual, with all of the popular datasets, there are *many* examples available online that can help you understand the data, clean the data, generate EDA plots to understand the distributions, etc. I **encourage** you to search for code to generate some reasonable plots with these data! (And, include all URLs that you use!)

The real focus of this lab is to learn how to use the scikit-learn framework to assist you with hyperparameter tuning. Almost every machine learning model has a number of parameters that you need to optimize for your given problem. The parameters usually come down to figuring out how to control the complexity of your model such that it is able to learn from your data, but not overfit. (Recall our bias vs. variance tradeoff discussion!) Understanding this is critical. Understanding how to address it is even better, and is an important skill to learn.

Generally, there are two approaches you can take to hyperparameter tuning. You could write an enormous number of loops yourself to track the model parameters you are evaluating, or you can use a grid search framework to search for the parameters. That's the purpose of scikit-learn's `GridSearchCV` class.

WARNING: **Hyperparameter tuning** can use a LOT of computation resources, both CPU and memory! Depending on how you set up your test, and the complexity of the model being evaluated, you could be waiting days for a result! So, ALWAYS start with a simple setup (small data sample, limited number of parameters to test) before you run an exhaustive grid search for your optimal parameters!

1) Read in the wine data frame using the code given above. It also gives you the names of the variables, which should align according to the data information given on the UCI web page.

2) Prepare your data. Minimally, you should be reporting:
   a) `shape`, `info()`, etc
   b) `describe()`
   c) Cast the type of the variables accordingly, particularly the target class
   d) Report on any missing data
   e) Show the `head()` of the data frame

3) Perform your EDA. Minimally, you should be reporting:
   a) Distributions of the target class
   b) Distributions of the variables
   c) Distributions of the variables, given each target class. Numeric and visual summaries are useful here.

4) Set up your `X` and `y` data frames to prepare for your modeling steps. Be sure to standardize your `X` variables. Create a binarized version of `y` as well. (Remember, this is important for classifiers such as neural nets when you are working with a target variable that is multi-class (i.e. more than two labels). And, again, it's very

important that you shuffle your data.

5) Let's induce a basic Decision Tree Classifier (i.e. `DecisionTreeClassifier()`). Don't set any initial parameters. Let the default tree induction parameters be used. Evaluate the tree using a 5-fold cross validation. Perform a standard report of your class-wide performance metrics (i.e. using classification report, but specify 3 significant digits instead of the default of 2). You should have all of this code completed from a previous lab.

   What is the overall accuracy? Which class performs the best? The worst?

6) Try to run your code above a few different times. Your performance results should fluctuate a bit. Why?

7) As we learned, a decision tree has a lot of parameters. Likewise, the DecisionTreeClassifier class offers those parameters for you to affect your tree induction.

   Let's introduce `GridSearchCV`, a fantastic framework to find good hyperparameters for your models. There are TWO important parameters to `GridSearchCV`:

   a) `estimator` – this is the model that you will evaluate, that implements the scikit-learn *estimator* interface (which is pretty much every model implemented in scikit-learn!)

   b) `param_grid` – this is a dictionary with parameters of your model as keys, and a list of values to test as the value of each key. Look at the page for `DecisionTreeClassifier`. For example: suppose we wanted to compare both measures of purity: "gini" and "entropy". And, suppose we wanted to compare different max_depth values. We could set up a param_grid parameter:

   ```
   param_grid = {
     'criterion' : ['gini', 'entropy'],
     'max_depth' : [3,4,5,6,7]
   }
   ```
   This represents 10 different runs (2 different criterion * 5 max_depth), times the size of your cross-validation.

   Add the above `param_grid` dictionary, then copy the following into your code, paying close attention to the parameters (i.e. just don't copy code without understanding it!):

   ```
   grid = GridSearchCV(DecisionTreeClassifier(), param_grid,
                       return_train_score=True, cv=5)
   grid_result = grid.fit(X,y)
   ```

   (If you did not use `X` and `y`, then you need to specify the variables you used here.)

   Execute your code. It should go ahead and evaluate *all* combinations of parameters you set up in your `param_grid`! Decision trees are pretty quick to learn, especially on small data sets such as this.

8) Cool! Now, you need to explore the internal attributes of `grid_result`. It contains a LOT of information. Look closely at the description of the return value on the documentation page for `GridSearchCV`. Then, show the best score, along with the best parameters for that score. (HINT `best_score_` and `best_params_`)

9) [M] What does the best "score" represent? What performance metric? How do you override it, if, for example, you want to use `f1_macro`?

10) Write the code to report the mean training and testing score and standard deviation for every parameter combination tested. Output your data in order of highest *mean test* score to lowest. How did you do compared to the default decision tree without any parameters? (HINT: This is easily done by casting the `cv_results_` parameter to a `DataFrame`, and then using standard pandas data selection and sorting methods.)

11) The default scoring parameter is accuracy, and you've already learned about how accuracy is not necessarily the best metric on unbalanced data. This data is only slightly unbalanced, thus it is not a bad metric. However, you should understand how to choose a better scoring metric to identify the best results for these data.

    In general, a macro-averaged f1 metric is a pretty good assessment of classifier performance when you have unbalanced data, as it will give you an honest assessment of how you are doing on the smallest classes.

    Redo the above steps, but you must accomplish three things:
    a) Add `min_samples_split` and `min_samples_leaf` criteria to your grid search. Choose the parameter values that make sense to you.
    b) Specify the `scoring` parameter to use *both* `accuracy` AND `f1_macro`. You'll likely also need to specify the `refit` parameter. (Look these up! Don't just blindly do what I'm telling you to do without understanding!)
    c) Change your result to a full 10-fold cross validation. This could take some time, but gives a better assessment of the power of your classifier. Perhaps using 5 fold just wasn't quite enough data for each member?

    Again, output your results, showing the top five configurations by `accuracy`, and also by `f1_macro`. Comment on your results.

12) [M] Discuss your findings. Did 10-fold make a difference? Did the additional parameters make a difference? Output the top 5 parameter configurations, along with the training and testing, sorted by accuracy, then sorted by f1_macro. Summarize.

13) Most of you should have some sort of multicore processor, even on the simplest laptops. You can easily take advantage of this. The default configuration of `GridSearchCV` is to NOT use any parallel processing, thus evaluating one model at a time. However, you could perhaps do 2 at a time! Do you have a CPU with more than 1 core? (Yes, you most likely do!)

    Redo the above, but try out the parameter `n_jobs=2` (I also sometimes set `pre_dispatch` to the same number as `n_jobs`, just to control too much memory usage. Not really important for decision trees, but could be important for larger models. If you have a really juicy machine, try an even higher value of `n_jobs`. (NOTE: Don't go higher than the number of cores in your machine as you will slow down quite a bit. Experiment by starting with only 2, and incrementally increase. NOTE: Parallel jobs for large models such as neural nets can sometimes fail with a memory error. If this happens, just run with `n_jobs=1`.)

    If you are able, do some basic time comparisons with different values of `n_jobs`, just to see how much time you save by parallelizing your grid search. If you set it too high, you will start to discover diminishing returns, and will harm your timing more then help, so back off the value. I typically do something simple:

```
import time
start_ts = time.time()
grid_result = grid.fit(X,y)
end_ts = time.time()
print("Time in seconds", end-ts - start_ts)
```

OK – You should notice that you were able to get better results by doing hyperparameter tuning (not to mention that an improved CV helped.) And, you should have noticed quite a boost in time by simply doubling the number of jobs you are completing simultaneously.

Depending on the number of parameters you choose, and the cv value, and most importantly, the model, this can take an *extraordinarily* long time! Like, seriously long. Like, if you are not careful, you could be wait days for results! Serious grid search for hyperparameter tuning is typically very CPU intensive (and memory intensive for large datasets.) So, ALWAYS START SIMPLE! Use only a few parameters to explore, with few values, and a small CV number to start with. With neural nets, choose a small number of epochs to begin. Why? **Confirm you are getting decent results first!** Then, let your obsessive side kick into high gear, set your system to evaluate many different parameters, and then typically, you go to bed. ☺

14) OK. Now on to hyperparameter tuning for Keras. (CONSIDER YOURSELF WARNED! Neural nets take a long time to train compared to decision trees!) From the previous lab, copy over your code for the function `create_keras_model()`. Now, modify your base neural net structure to match the inputs and outputs for the wine dataset you are working on for this lab.

    Modify the parameters of your function to allow you to pass model parameters of interest. In particular, you are going to want to evaluate a different `optimizer` and `activation` values for the hidden layer, and vary the number of hidden units. Modify your function to take these three additional parameters. You may name the parameters whatever you want, though generally it's good to keep them the same as the actual parameters used as you build your model.

    Once you have your new function, create your Keras classifier with `KerasClassifier()`. When you call this function, this is where you pass parameters such as `epochs` and `batch_size`. You don't want to wait long, so just use `epochs=5` and a `batch_size=4`. (I would advise starting with `verbose=1`, just to make sure your model is training properly, and you notice the accuracy increasing and loss decreasing, then set `verbose=0` when doing a bigger grid search on the next step.)

    Test out your `KerasClassifier()` instance with the `cross_val_predict()` method you learned about in the previous lab, and print the results of `classification_report`. You can try to vary some parameters if you want, but don't spend a lot of time here.

15) Use `GridSearchCV` on Keras (and prepare to wait several minutes!) First, create a `param_grid` dictionary. You must, minimally, specify the following keys:
    a) `'optimizer' : ['adam','sgd']`
    b) number of hidden units : 2 different numbers of your choice
    c) `'activation' : ['relu']`
    d) `'epochs' : [5, 10]`
    e) `'batch_size' : [4, 8]`

    Create a new classifier, but be sure verbose=0. Then, just like you did with the decision tree, wrap that classifier model in a `GridSearchCV` instance. Use a cross validation of 5 (though you can try more if you want.) Report your best model hyperparameters

    You should get some pretty good results!

16) Study your results. Can you do even better? How close can you get to 100% accuracy on a 5- or 10-fold cross validation with Keras? Try a different activation… say… `'tanh'`. Perhaps a few more hidden units? A bit larger batch size, but more epochs? See what happens….

Deliverables

Commit your **`lab12.ipynb`** file and push them to the remote Git repository. Be sure you have every cell run, and output generated. All plots should have `plt.show()` as the last command to ensure the plot is generated and viewable on Gitlab. DOUBLE CHECK GITLAB AFTER YOU PUSH! I only grade what I can view!