UDACITY

‹ Return to "Machine Learning Engineer Nanodegree" in the classroom

Plagiarism Detector

REVIEW

CODE REVIEW

HISTORY

Meets Specifications

Excellent work here with your dynamic programming and model development. Wish you the best of luck in your future!

All Required Files and Tests

The submission includes complete notebook files as `.ipynb`: "2_Plagiarism_Feature_Engineering" and "3_Training_a_Model". And the test and helper files are included: "problem_unittests.py", "helpers.py". The submission also includes a training directory `source_sklearn` OR `source_pytorch`.

All the needed files are included in the submission folder. You might also push your work to your github and share your github repo. This would also be a great way to showcase all your great work!

You can download an entire folder from AWS sagemaker to your laptop by opening a terminal on sagemaker. Navigating to the path where your folder is. Running the command to zip it

```
zip -r -X archive_name.zip folder_to_compress
```

You will find the zipped folder. You can then select it and download it.

All the unit tests in project have passed.

Looks good! All the unit tests in project have passed.

Notebook 2: DataFrame Pre-Processing

The function `numerical_dataframe` should be complete, reading in the original file_information.csv file and returning a DataFrame of information with a numerical `Category` column and new, `Class` column.

Nice work with your mapping! Using the `.loc[:,'Category']` approach is optimal with pandas.

There is no code requirement here, just make sure you run all required cells to create a `complete_df` that holds pre-processed file text data and `Datatype` information.

Notebook 2: Features Created

The function `calculate_containment` should be complete, taking in the necessary information and returning a single, normalized containment value for a given answer file.

Nice work computing your `calculate_containment` function. Very similar to how I would approach this with the `zip` command.

You might also create a secondary function to break up this code, so it can be re-used for other problems. Such as

```
def containment(ngram_array):
    ''' Containment is a measure of text similarity. It is the normalized,
       intersection of ngram word counts in two texts.
       :param ngram_array: an array of ngram counts for an answer and source text.
       :return: a normalized containment value.'''
    top = np.sum(np.amin(ngram_array, axis=0))
    bottom = np.sum(ngram_array[0])
    return top / bottom
```

Then call `containment(...)` in your `calculate_containment()` function.

Provide an answer to the question about containment feature calculation.

> "Calculation of the containment features is considered a pre-processing step which is to be done on both training and test data. We are not building a model yet so using the test data will not have any influence."

I would suggest also thinking about data leakage here as well, as there isn't any.

In essence, we only have the 'true' answer for one specific set of true questions. This is true for the training set, testing set, and all future data as well. This is considered to be the original 'truth' of the source text.

Maybe the best way to understand this, is if we have a 'brand new' question added. We would need to have some data points for both training and testing to validate the validity of the model on a new question. If we didn't do this, we would have no way to see how the model performs across all questions.

The function `lcs_norm_word` should be complete, taking in two texts and returning a single, normalized LCS value.

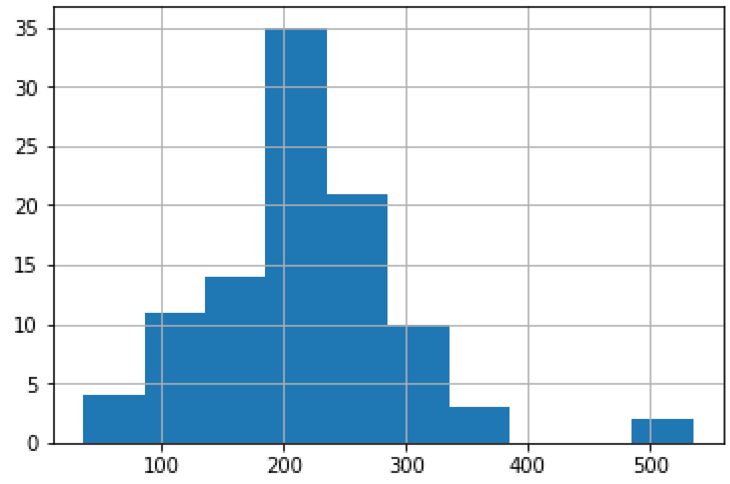
Excellent work with your dynamic programming code to generate the LCS value! Dynamic programming can be super useful to break down large problems into much smaller ones. One of my favorite dynamic programming problems is the Knapsack problem. If you want a good challenge, try taking a stab at this one as well.

You can also find many other problems similar to this on leetcode. This can really help you improve your algorithm game!

Define an n-gram range to calculate multiple containment features. Run the code to calculate one LCS feature, and create a DataFrame that holds all of these feature calculations.

Looks good! You might also check out the distribution of the length of the text fields as well.

```
# distribution of length of texts
import matplotlib.pyplot as plt
text_df.Text.str.split().map(len).hist()
```



Might it be reasonable to use a larger n-gram range based on the length of the answer texts?

Notebook 2: Train and Test Files Created

Complete the function `train_test_data`. This should return only a *selection* of training and test features, and corresponding class labels.

Your `train_test_data` function looks good! Nice one liner!

Select a few features to use in your final training and test data.

Provide an answer that describes why you chose your final features.

A threshold of 0.9 is a fine choice.

You might also check out this post for some more insight in why feature correlation matters (https://towardsdatascience.com/why-feature-correlation-matters-a-lot-847e8ba439c4)

Implement the `make_csv` function. The class labels for train/test data should be in the first column of the csv file; selected features in the rest of the columns. Run the rest of the cells to create `train.csv` and `test.csv` files.

Nice clean code! You might also think about running a `pd.DataFrame.dropna(axis=0)` to drop any incomplete rows. Typically is always good practice to run this and remove any incomplete rows.

Notebook 3: Data Upload

Upload the `train.csv` file to a specified directory in an S3 bucket.

Here might be a good link about creating an IAM Policy for Amazon SageMaker Notebooks (https://docs.aws.amazon.com/glue/latest/dg/create-sagemaker-notebook-policy.html)

Notebook 3: Training a Custom Model

Complete at least *one* of the `train.py` files by instantiating a model, and training it in the main if statement. If you are using a custom PyTorch model, you will have to complete the `model.py` file, as well (you do not have to do so if you choose to use an imported sklearn model).

Clean and simple. There really is no need for a pytorch model for this tiny dataset. No need for the additional overhead.

Define a custom sklearn OR PyTorch estimator by passing in the required arguments.

You might also have a command line argument for hyper-parameters

```
parser.add_argument('--C', type=float, default=1.0, help='regularization')
```

```
model = LinearSVC(random_state=100, C=args.C)
```

Fit your estimator (from the previous rubric item) to the training data you stored in S3.

Looks good! You might also look into doing some hyper- parameter tuning
https://docs.aws.amazon.com/sagemaker/latest/dg/automatic-model-tuning-how-it-works.html

One option for a smarter implementation of hyperparameter tuning is to combine random search and grid search:

- Use random search with a large hyperparameter grid
- Use the results of random search to build a focused hyperparameter grid around the best performing hyperparameter values.
- Run grid search on the reduced hyperparameter grid.
- Repeat grid search on more focused grids until maximum computational/time budget is exceeded.

Or could even look into using hyperopt. Here might be a good example of how to use this bayesian optimization technique in python.

Notebook 3: Deploying and Evaluating a Model

Deploy the model and create a `predictor` by specifying a deployment instance.

Nice work creating your endpoints. You might also find these links helpful for future PyTorch builds on SageMaker

(https://course.fast.ai/deployment_amzn_sagemaker.html)
(https://techblog.realtor.com/pytorch-aws-sagemaker/)

Pass test data to your deployed `predictor` and evaluate its performance by comparing its predictions to the true, class labels. Your model should get at least 90% test accuracy.

Provide an answer to the two model-related questions.

I always start with the simple model and also the data is quite small, so we can be worried about overfitting, so starting with a linear model to get a baseline score for the dataset is always a good idea.

> "A Linear Support Vector Classification is a very good choice because it produces clear 1 or 0 instead of probabilities like Logistic Regression."

I actually typically chose a Logistic Regression model over other linear models, since Logistic Regression often work fine with derived features from text data, I could check out the probabilities for each sample (as this might be useful, since if the model was around 40-60% confident, you could include some manual intervention, if desired). And lastly a Logistic Regression supports online learning. (although I would need to use create a pytorch model for this, or use a SGDClassifier model).

Notebook 3: Cleaning up Resources

Run the code to clean up your final model resources.

PLEASE keep this link handy to always remember to clean up your final model resources and save some money. (https://docs.aws.amazon.com/sagemaker/latest/dg/ex1-cleanup.html)

You might also even look into create a 'budget' on AWS, just to make sure you don't accidentally leave something running (https://console.aws.amazon.com/billing/home?region=us-west-2#/budgets)

DOWNLOAD PROJECT

RETURN TO PATH

Rate this review