UGA Data Science Competition Project Specification

Ayush Kumar

March 31, 2021

Contents

1 The Functional Approach

This is a document that is going to cover what we will be doing, and will attempt to divy up work. I will add as much detail as I can think of, but we will be using a functional approach to building the functionality for our project. Functional programming a paradigm that believes every function should be a pure function. A pure function is defined as a function that will return the same output when given an input, and will not modify the input in any way. It is difficult to implement a pure functional programming approach when it comes to data science due to inherent randomness in our models. Here are a couple things we will be borrowing from functional programming in our overall paradigm for this project:

- **Immutability** When building our project we need to make sure that when an input is passed into the function there it remains unchanged. This is to avoid unexpected behavior, and it can be very difficult in python, but packages like numpy are a good start.
- **Model Agnostic** When building non-model specific functions there should be no easy way to distinguish between what model is being inputted into the function, this will help us avoid redundancy
- **No Object Orientation** All non-model functions should not be implemented with an object-oriented approach. There should be no constructors in your code, and no instance variables. Every function should be self contained.

These concepts will help us parallel our coding, and help us move faster in this project. Using this approach it should theoretically be possible for us to develop code for multiple parts of the data science process, and not have to wait for one part to finish before we move on, but it requires very specific implementation. This document does not cover the explanatory portion because we are still researching explanatory approaches.

2 The Model Class

The Model will be the one object oriented part of the design for our project. Both of our models should follow the same overall structure, but there may be some key differences that will be discussed later on. We will have two classes: `log_reg()` and `feed_forward()`. We will be using other python packages, but this should help us streamline differences between scikit-learn and keras.

Here are the instance variables that both models will need:

- `model` - the Model
- `k: Integer` - the number of parameters in the model
- `n: Integer` - the number of observations in the training set
- `x: Numpy Array` - the input matrix, assumed to have no bias column
- `y: Numpy Vector` - the response variable

There are the functions that both models will need:

- `contructor(x, y)` - Will construct the model based on an input and response vector and assign all instance variables the correct value
- `predict(x = x) : numpy array` - Predicts the outcome of x based on the model, if no argument is used then training data is used to predict, should return raw probability not a 1 or 0

2.1 feed_forward

Additional Functions

- `train(x = self.x, y = self.y, ouput = True, epochs = 100, eta = 0.05)` - the train method should be used to train the model on training data that defaults the data passed on through the constructor.

3 Preprocessing

Since we will be working with multiple sources of data, we need an easy pipeline for making sure the data we are using will be in the same form no matter if we're using the training, validation, or testing dataset. We want this process to happen in one line, but it should be broken up into multiple functions:

- `intake_data(path: String): pandas.DataFrame` - takes in a csv as a pandas dataframe, we can just use pd.read_csv() for this step
- `impute(data: pandas.DataFrame): pandas.DataFrame` - imputes missing values using multiple linear regression
- `dummy(data: pandas.DataFrame): pandas.DataFrame` - turns categorical variables (states) into an appropriate number of dummy variables and removes all non-numeric variables
- `split(data: pandas.DataFrame, response: String): numpy.Array, numpy.Array` - takes a dataframe and splits it into the X data matrix and y response vector based on the name of the response vector

`read_data(path: String): numpy.Array, numpy.Array` - Does all 4 above steps and returns the arrays we need to input into the models

4 Hyper-parameter Tuning

Every hyper-parameter is tuned in a similar manner:

1. start with a value for the hyper-parameter
2. assess model performance with some sort of function (preferably directional)
3. adjust hyper-parameter using information from the loss function
4. stop adjusting at once a certain condition is met

4.1 Decision Rule

A large part of this project is using the predictions of a trained model in conjunction with the validation dataset to find an optimal decision rule for defaults (1) or not defaulting(0). We want to use a differentiable loss function to allow us to use a closed form solution or at very least use gradient descent to optimize this hyper-parameter automatically. The traditional decision rule assumes 0.5, but obviously we want to optimize this.

- `loss(y, yhat): float` - loss function based on actual decisions and predicted decisions based on the decision rule
- `optimize_lambda(predicted, actual): float` finds the optimal value for decision rule based on model output (between 0 and 1),
- `decision(predicted, lambda = 0.5): Integer` takes the predicted probability from the model, and converts it into a probability based on the decision rule (default 1 if greater than 0.5 and 0 if less than or equal to 0.5)

Possible loss functions that can be used: (Implement multiple and see which yields the best results)

- 0-1 Loss - if accurate prediction returns 0 otherwise 1 and all is added up, downside is that this is non-directional
- -1-0-1 Loss - if false positive (predicted 1 should be 0) -1, if correct 0, if false negative (predicted 0 should be 1) 1 sum over all observations
- Binary Cross Entropy - if you're up to the challenge, may be difficult, but some libraries have an implementation

4.2 Model Width

The Universal Approximation Theorem States that any function can be approximated by a 3-Layer Neural Network, since we don't have a crazy amount of input features we can treat the width of the hidden layer as a hyper-parameter to be optimized. Our loss function will be the difference in the accuracy scores of the training dataset, and the validation dataset. So long as the accuracy of the model is increasing we will continue to add to our model width, but once it begins to over fit the effect will be felt in the validation dataset, and this will serve as our stopping rule.

4.3 Regularization

Regularizing our Neural Network will serve the purpose of avoiding over-fitting, and we can use the same approach as the model width hyper-parameter to find the optimal regularization coefficient. We simply increase in a geometric fashion, so long as the distance in loss between the training and validations dataset is decreasing, once it stops decreasing we stop increasing the lambda. l2 regularization will be faster, but l1 will be better at avoiding over-fitting.

4.4 Learning Rate

Finding the optimal learning rate will help us speed up model development, and can help reduce the number of epochs needed for the model to converge.

4.5 Batch Size

Finding an optimal batch size will help speed up model development. The greater the batch size the fewer gradient step our network will have to take, but going to large can hamper the performance of the model. Finding an optimal batch size is key to reducing computational load.

5 Metrics & Evaluation

We need a few key metrics for analysis of our model:

- `confusion_mat(yhat: numpy.array, y: numpy.array)` - prints out a confusion matrix
- `accuracy(yhat: numpy.array, y: numpy.array)` - percent correct
- `percision(yhat: numpy.array, y: numpy.array)`
- `recall(yhat: numpy.array, y: numpy.array)`