**DS Grad Programme 6 (Machine Learning – Part Two) –   
Building and Maintaining Models**

The basics

* Model creation needs to exist in tandem with how the model will actually be used. Requirements would be different for each of the contexts below:
  + A web application for the general public on “.gov.uk”?
  + A one-off analysis for internal business forecasting?
  + An automated system deployed within our organisations servers for long term use?

Software development good practice

* Everything should follow RAP good practice guidance (duck book)
* Production grade models will be easier to implement if your experimental models follow good practice, as they will be easier to upgrade
* Documentation – Required examples (dependent on project need)
  + inline comments
  + function / class level document strings
  + library documentation
  + process diagrams
  + desk notes for other analysts
  + presentations for stakeholders
* Version Control
  + We need to use code version control to avoid losing previous iterations of the model design.
* Testing
  + By using automated code testing we can quickly assure that the code written meets a range of requirements, at both high (project level) and low (specific lines of code) levels.
  + This can be extremely important in modelling, helping us to answer key questions:
    - Does the complex data transformation we have written actually work?
    - Does the model predict when we would expect for specific examples?
    - What happens when incorrectly formatted data is presented to the model?
  + We can test for realistic situations that our software system may encounter, assuring us they will not fail incorrectly.

Reproducibility

Model persistence

* The idea of “saving” a model to some form of computer memory, such as a hard drive, is the idea of model persistence.
* Model persistence allows us to develop a model, then:
  + store it
  + share it
  + deploy it
* This way we do not need to retrain a model every time we want to use it (inefficient at best, impossible at worst).
* Broadly there are two ways to write a model to memory:
  + Using a programming language specific generic object format (for Python this would be a .pkl file, for R an .rds), useful if you know that the model will only be used using that programming language.
  + Using an inter-operable file format such as ONNX or Predictive Model Markup Language PMML which allow for development across different environments.

Model versioning

* When we write out a model to storage, we now “pin down” specific designs choices into permanent form. The design decisions and data used for that specific model are contained within what we have persisted.
  + By persisting the models we can access previous iterations of our design for later use. This will help us comparing what we have previously developed without retraining.
* As the number of models stored increases it becomes increasingly important that we keep track of what we have created. Each model created should be accompanied by relevant metadata, for example:
  + What code was used to train the model? (reference a version)
  + What data was used to train the model?
  + When was the model created?
  + Comments about the model, links to relevant documentation
* The metadata could be in a separate file (ideally located near the model!).
* How models created are going to be stored, recorded and documented needs to be decided before models start being persisted, or there will be an associated documentation debt build up.
* While there are some automated production approaches to this challenge, such as MLOps, for smaller scale projects, or those without the organisational data maturity to support advanced approaches, written documentation and solid naming conventions can go a long way to solving this.

Data versioning

* As versioning the model and code is important, the versioning of data is as well.
  + In a predictive modelling setting the model, and therefore the data are the products, and therefore need to be version controlled.
* Our training data can change over time, we may:
  + have new training data made available
  + alter the processing of our original training data
  + need to remove certain records in our training data (such as a result of a right of removal request)
* Two main options
  + One option is to use a versioning system, similar to git (used for code bases), called Data Version Control (DVC). When combined with an appropriate hosting platform this can streamline the process.
  + A more manual, but broadly applicable approach would be by naming the files used to train the data following an informative convention. By adding a data dictionary that gives information about each data set we can then keep track of changes the data more easily.

Experimentation and Evaluation Metrics

* Our experiment tracking should answer some of (but not limited to) the following questions:
  + What model was run?
  + How did the model perform?
  + When was it run?
  + What configuration of the model / processing was used?
* As with the persistence of models (saving the model to storage), the results of our experiments need to be permanent.
  + Therefore it is useful to have the model system produce its own results, such as writing evaluation scores, models and data versions out to a file.
* Two main options
  + At minimum, the experimental results we produce can be ‘written’ by hand, or automatically written out or “logged” to a file. This helps tie together the different elements of reproducibility.
  + More advanced approaches include other software systems that allow you to manage the tracking of experiments such as MLFlow, Tensorboard and guild.ai.

Delivery

Model Usage

* Considerations for deployment
  + We need to be able to answer the following questions when actually using a model, once it is created:
    - On what system will the model be stored? (on my laptop? on a server? in the cloud?)
    - Who will be responsible for monitoring it? (will the team that created it will be responsible for it’s use)
    - Who will interact with the model for it’s predictions? How will they give it data to make predictions for?
    - What happens to the predictions once they are made?
* Manual deployment
  + Process
    - The model is stored as a file on an analyst’s laptop.
    - Records for which predictions are needed are emailed over to the analyst.
    - The analyst loads up their code editor, loads the model onto their computer’s memory. They load in the data, clean it a bit then give it to the model for prediction.
    - The data and predictions are joined together and written out as predictions.csv and sent back to the person who requested the prediction. If there aren’t too many predictions, the analyst can look through the predictions to check whether anything has gone wrong.
    - Issues
      * Slow, and manual.
      * Time per prediction can be quite large considering the process is all computer based.
      * A batch can be processed, but if there are lots of batches or data, or a continuous stream this may be prohibitive.
      * Error prone
      * Dependent on an individual
    - Benefits
      * Does not require supporting IT infrastructure or operational knowledge
  + Automated deployment
    - Process
      * The model is stored on a server/cloud service.
      * The user goes to a specific website/address, completes some action causing the model to generate a prediction. The system records what the prediction was used. The user gets the prediction they wanted (or it happens to them).
      * An analyst can at some point look at the predictions made, exploring the results.
      * As the model is part of a software system the input data can be validated and rejected if needed. The predictions can be checked to make sure they meet certain expected conditions. All actions can be recorded as log data, ready for inspection later.
      * In this system we may be able to generate many more predictions at a time or over time than using an approach dependent on an individual.
    - Benefits
      * The system is as fast as the code we can optimise, rather than how quickly the analyst can reply to an email / run code.
      * We can automate checks for input data and output predictions.
      * We have an audit trail of data telling us what happened for each prediction / user, rather than relying on the analyst making notes.
    - Key Consideration
      * As with other pipeline building tasks, it should be considered how frequently a process will be used before creating a data infrastructure to support the process.
      * A one-off batch of predictions shouldn’t require a fully automated pipeline, but a model used every day does.

Monitoring of the model

* We cannot assume that the model will remain good forever in the future.
  + The model is modelling populations / dynamic processes that will change with time, they are likely not to be static.
* There are broadly 3 different things we want to monitor for a deployed model:
  + Stability - how stable is the data we are using to predict changing?
    - The distribution of predictions may change over time compared to the training data
    - The distribution and covariance of features may change over time when compared with the training data
  + Performance - how good is the model at predicting?
    - The value of our performance metrics (such as precision and recall) may be changing over time, we will need ground truth data to test this.
  + Operation - how is our system performing from a deployment perspective?
    - The model may be taking up lots of memory, or be overwhelmed with requests or even taking a significant amount of time to generate predictions.
* As we are using a software system, we can automate many of these processes. Some example tests we might continuously run are:
  + Check for predictions of NaNs or infinities
  + Check the average prediction bias is near zero
  + Check the predictions are within some reasonable size/magnitude

Concept Drift

* When the data we are using is changing its characteristics over time in comparison to the data we trained the model with, we refer to this as concept drift.
* This can cause major problems
  + If we are using a standard scaler in our model, that will have saved the mean and variance of our features. These characteristic values are now changing, so our scaling is going to be incorrect too!
* We can tell if there is concept drift by investigating the incoming data in batches.
  + If the sample at a specific time has significantly different characteristics to our training data we should raise a warning!
  + We can only tell if this concept drift is having an effect on performance if we have ground truth labels for the new data coming in.
    - This means we will need to have some way of verifying whether a new prediction is correct, even after our model is trained.
    - With the new ground truth data we can determine if our model is still fit for purpose, or if we need to change it in some way, the subject of the next section.

Retraining

* Stale models
  + As the underlying data and systems that the model is modelling change, the original model becomes stale. It’s performance will often degrade over time, so we need to mitigate against this performance degradation.
  + To appropriately maintain the model we need to:
    - Evaluate its changing performance (with new ground truth data)
    - Determine if the new performance is still good enough
    - If not good enough, update the model to meet our needs.
* Refreshing models
  + Option 1 – Retrain
    - Create a new training set with new data more representative of the current patterns (or with a mix of new and old data), and train a new model.
    - Model versioning becomes very important as we update the model we are using. We can roll-back to previous versions of models if needed.
    - We may even want to automate this retraining process, if our infrastructure is adequate and our quality assurance tests are robust enough.
  + Option 2 – Online models
    - Some classes of models can be trained on a set of data, then also trained on a second set, then third over time.
      * Models of this type are referred to as “online”, not necessarily because they are hosted online, but because they are always trainable. They can leverage prior training on new data.
    - Instead of “full” retraining, online algorithms are effectively “partially” trained at each step. This approach is often called “incremental” training or continuous learning.
      * Although we are updating an existing model, it’s important we keep versioning the models!
    - Some example online learning algorithms are:
      * Naïve Bayes
      * Stochastic Gradient Descent (SGD)
    - Good:
      * Easy to update model with new data
      * Efficient use of computer memory (not all data needed in memory at once)
    - Bad:
      * Only available for some learning algorithms
      * The algorithms available may not be the best performing
      * Changing design process will still require retraining
* Challenges of Retraining
  + There are a number of requirements we need to meet before we can do it:
    - We have an individual or system able to run the model training.
    - we have new data that is labelled appropriately and accurate.
    - We know when/at what point to retrain the model. This may be periodic, or only when performance reaches a certain level.
  + Some challenges in retraining a model include:
    - Collecting new labelled data
    - Knowing when to retrain the model
    - Having the expertise to automate retraining if appropriate
  + The best way to mitigate these challenges… is to plan for them!
    - The question of who is responsible for the deployment and/or maintenance of a modelling system is really key, the team needs to have the skills to support the system, regardless of how advanced it is.