Proposal Review Checklist

# Define Business Goal, Explore Data, Define Problem & Metric

* What exactly is the business problem to be solved?
* Is the data science solution formulated appropriately to solve this business problem?
* What business entity does an instance/example correspond to?
* Is the problem a supervised or unsupervised problem?
* If supervised, is a *target* variable defined? Think about the values it can take.
* Are the attributes defined precisely? Think about the values they can take.
* For supervised problems: will modeling this target variable improve the stated business problem? An important subproblem? If the latter, is the rest of the business problem addressed?
* Does framing the problem in terms of expected value help to structure the subtasks that need to be solved?
* If unsupervised, is there an “exploratory data analysis” path well defined?
* Is there a plan for domain-knowledge validation?
  + Will domain experts or stakeholders want to vet the model before deployment?
  + Will the model be in a form they can understand?
* Against what baselines will the results be compared?
  + Why do these make sense in the context of the actual problem to be solved?
  + Is there a plan to evaluate the baseline methods objectively as well?

# Implement Model

## Data Preparation

* Will it be practical to get values for attributes and create feature vectors, and put them into a single table?
* If not, is an alternative data format defined clearly and precisely? Is this considered in the later stages of the project?
* How exactly will the values for the target variable be acquired? Are there any costs involved? If so, are the costs considered in the proposal?
* Are the data being drawn from the similar population to which the model will be applied? If there are discrepancies, are the selection biases noted clearly? Is there a plan for how to compensate for them?

## Modeling

* Is the choice of model appropriate for the choice of target variable?
* Does the model/modeling technique meet the other requirements of the task? (Generalization performance, comprehensibility, speed of learning, speed of application, amount of data required, type of data, missing values)
* Is the choice of modeling technique compatible with prior knowledge of problem (e.g., is a linear model being proposed for a nonlinear problem)?
* Should various models be tried and compared (in evaluation)?
* For clustering, is there a similarity metric defined? Does it make sense for the business problem?

## Evaluation

* Is the evaluation setup and metric appropriate for the business task?
  + Are business costs and benefits considered?
  + For classification, how is a classification threshold chosen?
  + Are probability estimates used directly?
  + Is ranking more appropriate (e.g., for a fixed budget)?
  + For regression, how will you evaluate the quality of numeric predictions? Why is this the right way in the context of the problem?
* Does the evaluation use holdout data? Cross-validation is one technique.
* For clustering, how will the clustering be understood?

## Deployment

* Will deployment as planned actually (best) address the stated business problem?
* If the project expense must be justified to stakeholders, what is the plan to measure the final (deployed) business impact?

# Take Actions, Monitor Performance

# Appendix

## Setting up development and test sets

* Choose dev and test sets from a distribution that reflects what data you expect to get in the future and want to do well on. This may not be the same as your training data’s distribution.
* Choose dev and test sets from the same distribution if possible.
* Choose a single-number evaluation metric for your team to optimize. If there are multiple goals that you care about, consider combining them into a single formula (such as averaging multiple error metrics) or defining satisficing and optimizing metrics.
* Machine learning is a highly iterative process: You may try many dozens of ideas before finding one that you’re satisfied with.
* Having dev/test sets and a single-number evaluation metric helps you quickly evaluate algorithms, and therefore iterate faster.
* When starting out on a brand new application, try to establish dev/test sets and a metric quickly, say in less than a week. It might be okay to take longer on mature applications.
* The old heuristic of a 70%/30% train/test split does not apply for problems where you have a lot of data; the dev and test sets can be much less than 30% of the data.
* Your dev set should be large enough to detect meaningful changes in the accuracy of your algorithm, but not necessarily much larger. Your test set should be big enough to give you a confident estimate of the final performance of your system.
* If your dev set and metric are no longer pointing your team in the right direction, quickly change them: (i) If you had overfit the dev set, get more dev set data. (ii) If the actual distribution you care about is different from the dev/test set distribution, get new dev/test set data. (iii) If your metric is no longer measuring what is most important to you, change the metric.

## Basic error analysis

* When you start a new project, especially if it is in an area in which you are not an expert, it is hard to correctly guess the most promising directions.
* So don’t start off trying to design and build the perfect system. Instead build and train a basic system as quickly as possible—perhaps in a few days. Then use error analysis to help you identify the most promising directions and iteratively improve your algorithm from there.
* Carry out error analysis by manually examining ~100 dev set examples the algorithm misclassifies and counting the major categories of errors. Use this information to prioritize what types of errors to work on fixing.
* Consider splitting the dev set into an Eyeball dev set, which you will manually examine, and a Blackbox dev set, which you will not manually examine. If performance on the Eyeball dev set is much better than the Blackbox dev set, you have overfit the Eyeball dev set and should consider acquiring more data for it.
* The Eyeball dev set should be big enough so that your algorithm misclassifies enough examples for you to analyze. A Blackbox dev set of 1,000-10,000 examples is sufficient for many applications.
* If your dev set is not big enough to split this way, just use the entire dev set as an Eyeball dev set for manual error analysis, model selection, and hyperparameter tuning.

## Bias, Variance and Data Mismatch Errors

* Training set: This is the data that the algorithm will learn from. This does not have to be drawn from the same distribution as what we really care about (the dev/test set distribution).
* Training dev set: This data is drawn from the same distribution as the training set. This is usually smaller than the training set; it only needs to be large enough to evaluate and track the progress of our learning algorithm.
* Dev set: This is drawn from the same distribution as the test set, and it reflects the distribution of data that we ultimately care about doing well on.
* Test set: This is drawn from the same distribution as the dev set.
* Optimal error rate (“unavoidable bias”)​: the “unavoidable” part of a learning algorithm​’​s bias
* Avoidable bias​: the difference between the training error and the optimal error rate
* Variance​: the difference between the training-dev/dev error and the training error
* Data Mismatch: the difference between the training-dev and the dev-test error

## Techniques for reducing avoidable bias

* Increase the model size ​(such as number of neurons/layers): This technique reduces bias, since it should allow you to fit the training set better. If you find that this increases variance, then use regularization, which will usually eliminate the increase in variance.
* Modify input features based on insights from error analysis​: Say your error analysis inspires you to create additional features that help the algorithm eliminate a particular category of errors. These new features could help with both bias and variance. In theory, adding more features could increase the variance; but if you find this to be the case, then use regularization, which will usually eliminate the increase in variance.
* Reduce or eliminate regularization​ (L2 regularization, L1 regularization, dropout): This will reduce avoidable bias, but increase variance.
* Modify model architecture​ (such as neural network architecture) so that it is more suitable for your problem: This technique can affect both bias and variance.

One method that is **NOT** helpful:

* Add more training data​: This technique helps with variance problems, but it usually has no significant effect on bias.

## Techniques for reducing variance

* Add more training data​: This is the simplest and most reliable way to address variance, so long as you have access to significantly more data and enough computational power to process the data.
* Add regularization​ (L2 regularization, L1 regularization, dropout): This technique reduces variance but increases bias.
* Add early stopping​ (i.e., stop gradient descent early, based on dev set error): This technique reduces variance but increases bias. Early stopping behaves a lot like regularization methods, and some authors call it a regularization technique.
* Feature selection to decrease number/type of input features:​ This technique might help with variance problems, but it might also increase bias. Reducing the number of features slightly (say going from 1,000 features to 900) is unlikely to have a huge effect on bias. Reducing it significantly (say going from 1,000 features to 100—a 10x reduction) is more likely to have a significant effect, so long as you are not excluding too many useful features. In modern deep learning, when data is plentiful, there has been a shift away from feature selection, and we are now more likely to give all the features we have to the algorithm and let the algorithm sort out which ones to use based on the data. But when your training set is small, feature selection can be very useful.
* Decrease the model size ​(such as number of neurons/layers): ​Use with caution.​ This technique could decrease variance, while possibly increasing bias. The advantage of reducing the model size is reducing your computational cost and thus speeding up how quickly you can train models. If speeding up model training is useful, then by all means consider decreasing the model size. But if your goal is to reduce variance, and you are not concerned about the computational cost, consider adding regularization instead.
* Modify input features based on insights from error analysis​: Say your error analysis inspires you to create additional features that help the algorithm to eliminate a particular category of errors. These new features could help with both bias and variance. Adding more features could increase the variance; but if you find this to be the case, then use regularization, which will usually eliminate the increase in variance.
* Modify model architecture​ (such as neural network architecture) so that it is more suitable for your problem: This technique can affect both bias and variance.

## Addressing data mismatch

* Try to understand what properties of the data differ between the
* training and the dev set distributions.
* Try to find more training data that better matches the dev set examples that your algorithm has trouble with

## Learning curve (Error vs.. Training Data Size)

