PRELAB 5

INTRODUCTION TO MODEL SELECTION

OUT-OF-SAMPLE VALIDATION: when we evaluate our models we use data that wasn't used in model training

- Important fundamental principle
- Important technique used for evaluating a model's ability to generalize to new data.
- When our modeling goal is for GENERALIZATION, we can only trust the metrics computed on non-training data.
  - We care that our model performs well on past events only if that performance translates to future event performance. ← GENERALIZATION
- We care about out-of-sample validation because we can't fully trust our training data.
  - Assume we have an oracle that creates our data (DATA GENERATING PROCESS)
    The error or loss we want to optimize is on the theoretical data set & we call this the EXPECTED LOSS.
  - EX) ML engineer at Netflix; the business model depends on subscriptions ∴ an important modeling task is to determine whose likely to keep their membership for at least a year.

    Theoretical data universe: potential customers in the world with emphasis on future customers

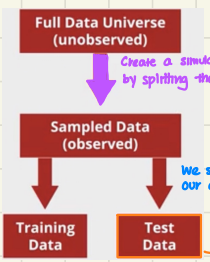
    ⇓

    But we can't observe these future customers at the time of model development so our observed data that we can use for training only includes past customers which is a subset of the full data universe.

Due to the inherent randomness in the data, the loss we often observe in training will be different and a little optimistic than from the expected loss.

This discrepency between expected loss and training loss is one of the biggest challenges we'll face when developing models.

- **EXPECTED LOSS**: The error or loss we want to optimize again on the theoretical data.
- To solve this problem:



Approximation of the full data universe
We cannot compute expected loss because it's THEORETICAL

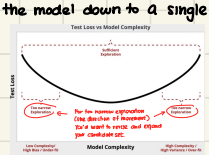
Highly likely that the training loss will be different from the test loss.

This will happen from either OVERFITTING or UNDERFITTING on the training data.

OUR TRUE GOAL is to optimize our OUT-OF-SAMPLE LOSS and we do this through a process called MODEL SELECTION

MODEL SELECTION: The process of MODEL SELECTION aims to search this base of model design options to find the perfect balance between over or underfitting

- When we hit that spot we'll end up with a model that optimizes the expected loss which means it'll generalize well on future data.
- Selecting the best performance model is strictly an EMPIRICAL PROCESS. — meaning we're not using any formulas or theories to design the best model in advance.
- We choose a set of candidate models, loop through them all, and select the one that best meets our performance criteria which is usually some optimal measure of loss on our test set.
- 2 criteria for successful model validation:
  1. Identify a good set of candidates to try out. ← MORE SUBJECTIVE ∴ knowledge and intuition becomes your guide.
     - If your search isn't varied enough you may miss out on some good candidates.
     - When designing appropriate candidate set, the most important thing to consider is the DIVERSITY of options when it comes to model complexity.
       - There should be a middle ground where the model is flexible enough to learn subtle patterns in the data BUT isn't so flexible that it overfits.
       - Should be able to identify candidates that span the full range.
     - CHALLENGE: can't reduce the model down to a single complexity score ∴ we'll use SUBJECTIVE JUDGEMENT
     - If we produce a plot like:



We'll feel more confident that we've chosen a sufficiently broad set of model candidates.

↳ MODEL DESIGN DIMENSIONS:

- Algorithm (i.e. LogisticRegression, KNN, Decision Tree, etc.)
- Features (which subset from full available set)
- Hyperparameters (broad range specific to each algorithm)

← Each model candidate will be specified by a single algorithm type, a single subset of features, and a set of hyperparameters that are specific to the given algorithm.

EX)

```
model_candidates = [
    DecisionTreeClassifier(max_depth = 4, min_samples_leaf = 1024),
    DecisionTreeClassifier(max_depth = 16, min_samples_leaf = 128),
    LogisticRegression(C = 0.001),
    LogisticRegression(C = 1000)
]
```

2 algorithms BUT each with different hyperparameter configurations

Algorithm + Hyperparameter config
Shows 4 examples of an algorithm and specific set of hyperparameters chosen for that algorithm.

```
feature_sets = [
    ['X1', 'X2', 'X3'],
    ['X1', 'X2', 'X3', 'X4', 'X5', 'X6']
]
```

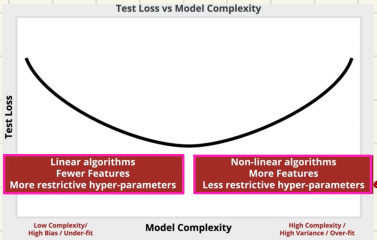
Feature config
2 candidate feature sets. We're either using all of it or half of it.

```
for candidate in model_candidates:
    for X_set in feature_sets:

        candidate.fit(df[X_set], df[y])
```

Algorithm + Hyperparameter + Feature set
Loop through the model candidates and the feature sets.
We call the fit method to build a model predicting our theoretical label Y.
Each iteration gives our full model specification along 3 design dimensions

↳ How our candidate selection fits into the model complexity vs. test loss curve:

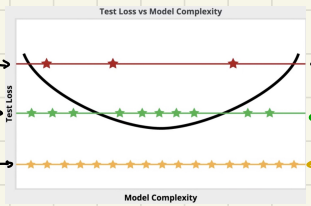


- Linear models tend to be lower in complexity.
- KNN and decision trees are very flexible & can have high complexity
- More features gives the algorithm more opportunities to fit the data, which is how we define model complexity
- HYPERPARAMETERS restrict or enable the flexibility of the given algorithm.

← Each dimensions will enable us to span the low vs. high complexity range.

↳ When designing candidate sets we need to determine HOW MANY candidates is enough.

- Use SUBJECTIVE JUDGEMENT and find the MIDDLE GROUND between 2 extremes.



3 different sets of model candidates →

← Only evaluate 3 different candidates
The range is NOT SUFFICENT enough to get close to the optimal point

← The compromise between RED and YELLOW.
Has enough candidates to cover full range of model complexity but has half the training cost.

← YELLOW doesn't have the problem RED has.
Has lots of candidates so we probably do best at finding the optimal point. BUT it comes at a cost. Each point here requires training a model & depending on the data size it can be an expensive process.

The choice of how many candidates models to test ultimately becomes a tradeoff between training time & test set performance.

② Apply an appropriate out-of-sample evaluation. ← IMPORTANT TO GET RIGHT

↳ Methods implemented in scikit-learn:

Search through set of model candidates to find the right level of model complexity.

PERFORMING OUT-OF-SAMPLE VALIDATION

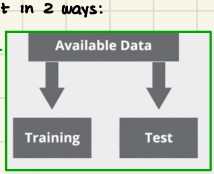
Best way to achieve generalization is to evaluate our loss function on out-of-sample data to perform MODEL SELECTION

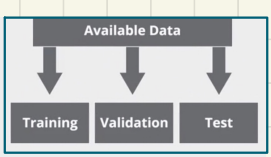
CREATING OUT-OF-SAMPLE DATA:

- Take available data and split it in 2 ways:

Don't need to do model selection

We skip the validation step





← Split the data into 3 sets
This is how we do it if we want to perform MODEL SELECTION.

We then select the best performing option from the validation sets, best performer becomes our final candidate model.

After that we evaluate the best performer on the test set to estimate how our model performs on the data.

Training set = used to actually fit the model to the data

Validation set = used to evaluate model candidates for model selection

Test set = used for estimating the generalization performance of the best selected model

← Ex. Assume we have 10 model candidates we'll train on the data 10 times & have 10 separate scores scored against the validation data.

IMPORTANT RULES OF TEST SET USAGE:

1. Test set needs to be representative of the data the model will be applied to.
   EX) If our model is going to be used in some future events, we'll want to use test data that has our representation of those future events.
2. Test set should be independent from training (no overlap in units)
   If this assumption is violated in any way then we're at risk for being too optimistic about our model's performance.
3. No model selection or training should be done after evaluating on test set.
   Test set should be used exclusively for estimating future performance of the model & not for any decisions related to model selection or model fitting.

CREATING TRAINING, VALIDATION, AND TEST SETS

2 PRINCIPLES

- When performing out-of-sample validation we need to make sure that the data we use is completely INDEPENDENT from the training data & it's representative of the data the model will be applied to.
- Both rules help ensure the out-of-sample data will not OVERFIT model & the model will generalize to new previously data.
- The 2 principles can be achieved with PROPER SAMPLING STRATEGY.
- To determine which sampling strategy always ask 2 key questions about your data:
  1) Do the units of analysis appear multiple times in the data?
  2) Is there a time dimension?

EX)

UserID	TimeStamp	X_1	X_2
AAA111	1	0.36	0.32
AGH222	1	0.07	0.17
AAD555	1	0.98	0.82

Q1: No
Q2: No

⇓ When the answer to both of these questions is NO

We can just pull a simple random selection of all examples into a table set.

HOW TO DO THIS ON A DATAFRAME

```
sample_pct = 0.2 #what % of data should we use for out-of-sample

n = df.shape[0] #get number of rows in our data

randomizer = np.random.random(n) #generate a random sequence of size n

train = df[(randomizer > sample_pct)] #generate training data on 80% of data
test = df[(randomizer <= sample_pct)] #generate testing data on 20% of data
```

← specify how much data we want to split in a test set

Generate random sequence to use as a filter

Use the common DF filtering logic to create and train a test set.

EX)

UserID	TimeStamp	X_1	X_2
AAA111	1	0.84	0.75
AAA111	2	0.07	0.92
AAD555	1	0.12	0.48
AAD555	2	0.32	0.25
AGH222	1	0.22	0.82
AGH222	2	0.24	0.63

same unit ID but different time stamp

Q1: Yes
Q2: Yes

⇓ We can't randomly sample the examples into the training & test set if we did, we would have examples from the same unit in both the training and out-of-sample datasets. This would violate the independence rule

We need to randomize at the unit level

Example implementation of this strategy:

We used a combination of hashing and modulant fuctions to match each record to a singular bucket.

The bucket function is deterministic meaning that if we put the same input then we'll always get the same output.

```
def id_hasher(x, buckets = 100):

    return (hash(x) % buckets) / float(buckets)

df['hash_bucket'] = df['id'].apply(id_hasher) #map record to a sampling bucket

sample_pct = 0.2
train = df[(df.hash_bucket > sample_pct)] #generate training data on 80% of data
test = df[(df.hash_bucket <= sample_pct)] #generate testing data on 20% of data
```

	id	x	hash_bucket
0	a	1	0.72
1	a	2	0.72
2	b	3	0.05
3	b	4	0.05
4	c	5	0.56
5	c	6	0.56
6	d	7	0.29
7	d	8	0.29

When the data has a time dimension, we want the latest data to be our test & validation data.

UserID	TimeStamp	X_1	X_2
AAA111	1	0.42	0.75
AGH222	1	0.06	0.28
AAD555	1	0.04	0.92
AZZ121	2	0.98	0.64
ALP090	2	0.57	0.83
AMM655	2	0.26	0.39

Usually latest data we have serve as the best proxy for future data

When answering yes we would want to apply to both strategies:

1. Randomize the units
   ↳ with each unit assigned to separate buckets.
2. Take latest example of those units assigned to the out-of-sample test data sets.

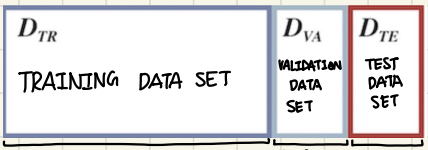
SPLITTING DATA SETS FOR MODEL SELECTION

OUT-OF-SAMPLE VALIDATION: using data that a model was not trained on to evaluate the model.

↳ Performed to properly evaluate how well a model generalizes to new, previously unseen data.

↳ Performing model selection to choose an optimal model for our data and predictive modeling problem, we split the initial data set D into 3 mutually exclusive subsets:

NOTE: When performing model selection we should split data set to TRAINING, VALIDATION, & TEST SETS to properly evaluate model performance & prevent overfitting.



Common choice is to split the data: 70/20/10

HOW TO SPLIT A DATA SET:

- All 3 sets should be drawn from the same data distribution.
  - ↳ THUMB RULE: The data sets simulate the real life settings in which the algorithm will operate.
- COMMON METHOD to split data into different sets: UNIFORMLY AT RANDOM
- IF DATA HAS TEMPORAL COMPONENT & CHANGES OVER TIME then you should split the data BY TIME & make sure that you never predict the past from the future & always the future from the past.

  EX) For temporal component: email spam

  Spam mails change over time as spammers adapt their emails to get past spam filters

  To address this problem we can collect data for a few weeks to train your model. Then, collect data for a few days to use as your D_{VA} data set to validate your model. Finally, collect data for a few more days to generate the D_{TE} data set to test your model.

WHY IS VALIDATION DATA IMPORTANT:

- Common scenario is that the 1st ML algorithm that's trained on D_{TR} does not perform well on D_{TE} and needs further refinement.

  EX) Email spam filter with objective to catch 99% of all spam, with at most 1/1000 false positives

  ↳ If initial algorithm is not accurate enough for your needs, you need to either CHANGE your ALGORITHM or its HYPERPARAMETERS.

  PROBLEM: cannot tweak your algorithm to perform well on D_{TE} or else you'll be overfitting your model & not improve its accuracy on new data.

  ↳ The error obtained on D_{TE} is only an unbiased estimate of the true generalization error of the model if the model were trained independently of this test set. The moment you make changes to the algorithm it's no longer independent. This is where validation data set comes in.

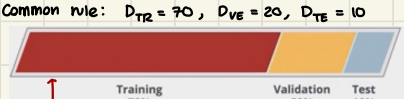
The validation data set is a proxy for the test set. In practice, you train your algorithm on the training set and evaluate it on the validation set. If your model is not satisfactorily accurate, you continue tweaking it until the validation error improves to an acceptable level. You then complete a single and final evaluation of your model on the test set D_{TE} to find the unbiased estimate of the generalization error of your final model. Often, the final model is re-trained on the union of D_{TR} and D_{VA} so as to not waste the examples in the validation set.

CROSS VALIDATION

Cover what the sizes of the training, validation, and test data sets be.

Cover what happens if the standards splitting procedures don't give you enough data for training, validation, and test.

- Common rule: $D_{TR} = 70$, $D_{VE} = 20$, $D_{TE} = 10$



Training usually involves the MOST data.

∴ majority come here.

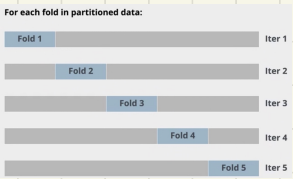
In all 3 data sets, the sample size affects the variance of what you're estimating.

∴ More data = less error

∴ want sufficient data in each bucket ∴ compromising one bucket to benefit the others.

- Cases where there's not ENOUGH of data: implement **K FOLD CROSS EXAMINATION**.
  - **K FOLD CROSS VALIDATION**: resampling method that uses different portions of the data to train and validate the model on different partitions of the data.
    - Allows us to **RECYCLE the data** so that we can use every data point for both training & validation.
    - STARTS with first reserving a small subset of the OG data to use as your test set
      NEXT take the remaining data & split it to k-equal size partitions which we call **FOLDS**

Iterate through each fold to create training & validation split



With each iteration we split the data so the fold associated with the iteration number becomes the validation data.

Remaining data is the training data.

Within a given iteration we then apply our standard procedure which is to fit a model to the training data & evaluate it against the validation data.

Each iteration will produce a loss from the validation set.

Once loop is done we would have k-separate measures, each taken from a separate partition of the data used as validation.

VALIDATION LOSS is the average of k-separate estimates.

K-FOLD-LOSS = average of validation loss across k folds.

- Question to consider: HOW MUCH FOLDS TO USE
  - Most common number is 10 folds (has lots of empirical validation)
  - REMEMBER: more folds = more time for training

WHOLE PROCEDURE IN CODE:

Manages the k-Fold process

```
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score

folds = KFold(n_splits = 5)

val_folds = []

for train_index, val_index in folds.split(df):

    df_train = df.iloc[train_index]
    df_val = df.iloc[val_index]

    model = DecisionTreeClassifier(max_depth = 32, min_samples_leaf = 50)
    model.fit(df_train[X], df_train[y])

    preds = model.predict(df_val[X])

    val_folds.append(accuracy_score(df_val[y], preds))

print("Fold Accuracies = {}".format(val_folds))
print('')
print("Average Fold Accuracy = {}".format(np.mean(val_folds)))
```

Define our faults using the k-fold classroom scikit-learn

Loop through the procedures using the k-folds split methods This help creates indexes for us that we can use to explicitly split the data into train & validation sets.

We then can estimate the model and get the validation loss

```
Fold Accuracies = [0.04754139488208731, 0.03487205218263924, 0.019693928750627195, 0.05381334671349724, 0.03148914816
208757]

Average Fold Accuracy = 0.0374819741381877O6
```

When we need to compare a different model configurations from our candidate set

OUT OF SAMPLE VALIDATION TECHNIQUES

Splitting our data into different subsets can help us properly evaluate whether our model generalizes well to new data.

2 common methods:

1) **HOLDOUT METHOD**:
   - Randomly splits a data set into training and validation subsets.
   - Trains on the training set.
   - Tests our model's performance on validation set.
   - Typically used when we have LARGE data.

2) **K-FOLD CROSS-VALIDATION METHOD**:
   - Splits the data set into equally sized subsets or FOLDS with k representing the # of folds.
   - Perform HOLDOUT method (train + validation) k times, each time we train on k-1 training folds and test on one validation fold ∴ each fold will have a chance to serve as a validation set.
   - Then average the resulting prediction accuracies obtained on each of the k iterations to get a good estimate of our model's performance on new data.

EX)

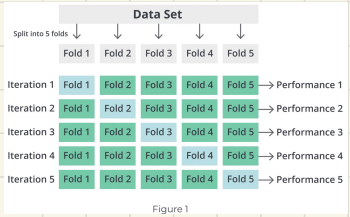


Figure 1

1. Split initial data set into 5 equally size folds
2. At each iteration we train on 4 GREEN FOLDS and validate on 1 BLUE FOLD. We continue this training and validation process in the round-robin fashion until all 5 folds had a chance to be the validation set.
3. Average the resulting prediction accuracies obtained on each of the 5 iterations to estimate how effective our model is.

MODEL SELECTION DEMONSTRATION: USING K-FOLD CROSS VALIDATION

Supervised learning task

STEP 1: Choose our model candidates

- We'll use Decision Tree classifiers as our algorithm
- We'll consider the hyperparameter: max depth
  - ↳ To control the complexity
- We'll use all available features in the data.

RANDOM STATE because when we do model selection with cross validation we need to use the same fill definitions for each model candidate to ensure our comparisons are on the same data.

```
folds = KFold(n_splits = 5, shuffle = True, random_state = 10)

model_scores = []
model_parms = []

max_depths = [2**i for i in range(10)] #set up range for max_depth grid search

for md in max_depths: #loop through model candidates (i.e. hyper-parameters)

    model = DecisionTreeClassifier(max_depth = md)
    score = run_cross_validation(df, X, y, model, folds) #returns avg score across folds

    model_scores.append(score)
    model_parms.append(md)

score_df = pd.DataFrame({'score':model_scores, 'parm':model_parms}) #put results into a dataframe
```

Initialize the K-Fold

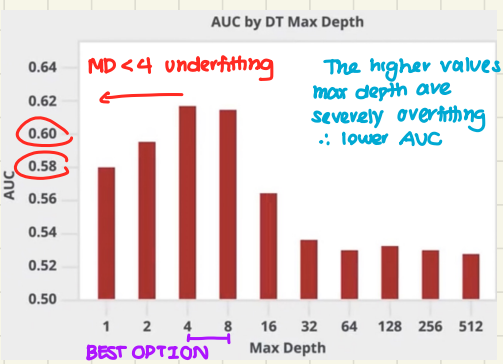
Since we're searching over max-depth parameter we define range using increasing power of 2

Initialize model candidate

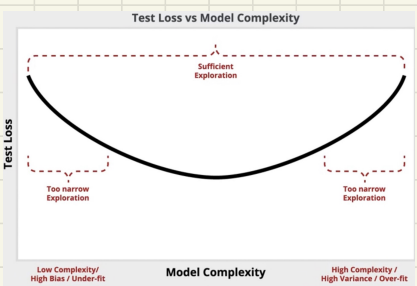
pass it into the cross validation function This function returns a METRIC using the metric AUC which is a classification metric we want to maximize.

After looping through each candidate we store the result of each loop into the DF

RESULTS OF CODE PLOTTED:



NOTICE... plot shows the same shape shown previously when plotting model complexity vs. expected loss



The shape of this graph is inverted because we chose an evaluation metric which maximizes as oppose to minimizes.

SCIKIT LEARN'S PROCEDURE IN FEW LINES:

```
max_depths = [2**i for i in range(10)] #set up range for max_depth grid search

folds = KFold(n_splits = 5, shuffle = True, random_state = 10) #define folds

clf = GridSearchCV(DecisionTreeClassifier(),
                   param_grid={'max_depth':max_depths},
                   cv = folds,
                   scoring = 'roc_auc'
                  )

clf.fit(df[X], df[y])

print(clf.best_estimator_)
```

```
DecisionTreeClassifier(max_depth=4)
```

GRID SEARCH: we specify grid of hyperparameters and we're searching through different combinations of them.

Performs cross validated grid search for you

It allows you to search through a range of hyperparameter options that you've specified.

MODEL SELECTION

MODEL SELECTION: process where we choose 1 final model from among many candidates that results in the optimal model for a given data set and ML problem.

↳ Choosing an optimal ML model involves:

① Performing **out-of-sample validation** so we can properly evaluate how our model generalizes to new, previously unseen data.
Split our data into **training**, **validation**, & **test** data sets.

- **training**: used for model fitting
- **validation**: perform model selection (e.g. after evaluating the model's performance we tweak the model's hyperparameter configurations)
- **test**: For the final evaluation of the chosen model's performance

② Performing **FEATURE SELECTION** so we can reduce the number of features to only use relevant data in the training of our model. This will improve a model's performance.

③ Determining the **OPTIMAL HYPERPARAMETER CONFIGURATIONS** that results in a well-performing model.

Choosing hyperparameter configurations:

Choosing the best hyperparameter value is an EMPIRICAL PROCESS.

Searching Methods to pinpoint the optimal sets of hyperparameters for maximizing model performance:

① **GRID SEARCH**: goes through various combinations of hyperparameters systematically (i.e. **geometric progression** or **exponential progression**)
This ensures good coverage within some regions determined by prior knowledge or heuristics.

② **RANDOM SEARCH**: Instead of choosing our hyperparameter values systematically, we'll be selecting the hyperparameter values at random in order to capture a wide variety of combinations that could otherwise be missed by being systemic about choosing our hyperparameter.

GUEST SPEAKER: CHOOSING THE BEST MODEL FOR YOUR ML PROBLEM

NO FREE LUNCH THEOREM: no one algorithm works well for every problem ∴ we should try many diff algorithms for your problem while using a hold out test set of data to evaluate performance and select a winner model.

Factors affecting the choice of the models are:

1. Does the model meet the business goals?
2. How much data preprocessing does the model require?
3. How accurate is the model?
4. How transparent is the model?
   Is the model easy to understand?
5. How much time is required to build the model?
   How fast can it make predictions?
6. Is the model scalable once deployed to production?
   ↳ How efficiently it can handle high-dimensional data w/ high dimensional data w/ large number of features & examples & instances.

FEATURE SELECTION

FEATURE SELECTION: process of empirically testing different combinations of features to choose an appropriate set.

↳ **WHY DO FEATURE SELECTION?**

- **Less overfitting**: some features have little to no predictive power ∴ we want to remove these features
- **Better interpretability**: fewer features makes the model easier to explain
- **Better scalability**: consider scalability & stability of predictive system.
- **Lower maintenance cost**: Every feature you use incurs runtime cost, development costs, and maintenance cost.

↳ FEATURE SELECTION METHODS:

④ **HEURISTIC SELECTION**: create some rule/heuristics to filter out features using heuristic rules prior to modeling.

NOTE: Can be used in Logistic Regression we just don't need to.

↳ Provides balance between scalability, simplicity, and efficacy.
↳ called heuristics because we're not going to use traditional model selection techniques & as our guide.
↳ We'll design a rule before any modeling is done & we'll use that as our basis for selection.
↳ **RULES USED TO FILTER FEATURES PRIOR TO MODELING:**

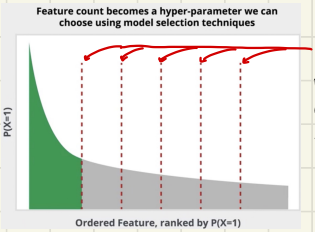
1. Feature has a minimum level of correlation or mutual information with the label
   - Considers the relationship between the label and the feature
   - If we can't detect a relationship we remove the feature

2. Feature has sufficient SUPPORT (i.e., % of examples where feature is not 0/NULL)
   - 0/null are cases that don't add predictive power to the model
   - When support of a feature is low, we won't expect the feature to perform well in the final model
   - Highly scalable & is useful when dealing with large amount of features.

EX) When data comes in the form of text, common feature engineering is to convert each word to binary feature.
1 = a particular word is in the document / example
0 = otherwise

EX) Tweets
We can compute the SUPPORT of each word feature
↳ small set of words == medium-high support
↳ most words == low support == occur rarely in the data

- CHALLENGE: introduced decision parameter which is the exact threshold we need to choose for feature inclusion.



Different thresholds we can choose for feature selection

We can decide this threshold by just picking one and sticking to it.

OR

Take an empirical learning approach

THRESHOLD (# of features): can become a hyperparameter that we can ultimately select using model selection techniques

We can set up an experiment where we start with the leftmost threshold & run cross validation.

Then iteratively increase the threshold & choose the value that gives the best cross validation performance.

3. Domain specific rules (i.e., feature is too expensive to operate, feature not allowed by regulations)
   - Don't invest efforts in building features you know you can't use.

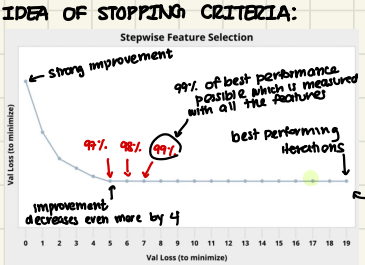
② STEPWISE SELECTION: iteratively add/reduce features based on empirical model performance (usually done in increment fashion)

↳ ALGORITHM IN PSEUDOCODE:

1. Initialize: best_subset = {}
2. Initialize: candiate_features = all features ← potential features we're skipping
3. For each feature in candidate_features:
   a. Get: (cross) validation score with model built with: best_subset + feature ← can use either cross validation or regular validation
   b. Add to list: (feature, (cross) validated score)
4. Choose: best_feature = feature from step (3) with best performance
5. Update: best_subset = best_subset + best_feature
6. Remove: best_feature from candidate_features
7. Repeat: steps 3 - 6 until stopping criteria is met

↳ Can be used in Logistic Regression we just don't need to.

↳ IDEA OF STOPPING CRITERIA:



Each point represents validation loss after training a model on the best subset up until that stopping point

NOTE: Depending on metric we choose, we want to either minimize/maximize our loss.

← we're minimizing our metric loss

- To define stopping criteria is to stop when the relative improvement after each iteration is below some level.

Stopping criteria = $Val(t)/Val(t-1) - 1 < \epsilon$

↑ Epsilon we choose will be subjective decision we'll make but usually low % improvement like 1% works.

③ REGULARIZATION: include penalties for feature count in the algorithm's loss function.

↳ Changes the loss functions of the algorithm to incur extra loss for each feature we use.
↳ Each feature needs to add material value to the model to compensate for the extra penalty.
↳ Allows us to reduce the future count within the model training process
We'd still have to run model selection procedures to choose relevant hyperparameters associated with regularization but we don't have to run feature selection methods.
↳ IMPLICIT FEATURE SELECTION: reducing feature count as a byproduct of the model training procedure.
   - Regularization performs IMPLICIT FEATURE SELECTION by modifying the loss function we use for training
   - Only use regularization with specific set of learning algorithms: Logistic & Linear regression
     DOESN'T APPLY TO: KNN & DT

C controls the weight of the penalty. Higher C = less regularization

Regularization Loss = Loss + (1 / C) * Penalty

Loss can be any common loss function, such as Log-Loss or MSE

we redefine our loss function for LINEAR or LOGISTIC REGRESSION by adding PENALTY.

→ **PENALTY**: related to the # of features you have in your model and their particular weights

→ More features == larger the penalty

→ Creates a TRADEOFF we'll need to manage: modulated by the parameter C.

- C is large == features incur less penalty == no contribute to high reduction in the loss component to be included.

→ **2 PENALTIES**: 2 VARIANTS:

$$L1 - Penalty = \frac{1}{C}\sum_{j=0}^{m}|w_j|$$

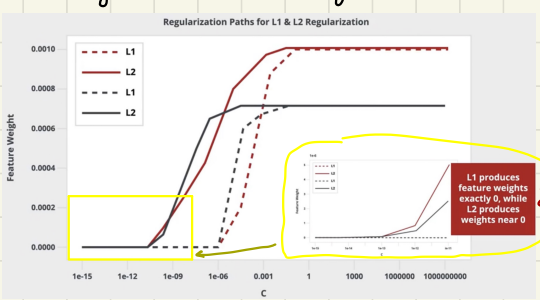
a.k.a., the "Lasso"

$$L2 - Penalty = \frac{1}{C}\sum_{j=0}^{m}|w_j^2|$$

a.k.a., "Ridge Regression"

Both cause the weights that the model learns to get smaller because larger weights incur higher penalties.

→ Effect of regularization on model's weights:



- As C ↑ the regularization penalty ↓ ∴ feature weights can be larger
- There's an asymtope where increasing C will eventually have no effect.

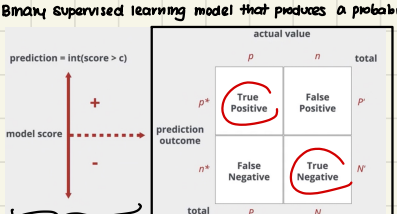
This is where implicit feature selection take place

When penalty is high

→ After running the hyperparameter selection to identify the best value of C we can review which features has the weights 0 or ~0. We can then remove these features from our data b/c the resulting model will no longer use them.

CONFUSION MATRIX & CLASSIFICATION METRICS

- Binary supervised learning model that produces a probability score like a logistic regression or DT.



we assign a particular example to a class by determining which side of a given threshold the example scores lie.

four different scenarios created.

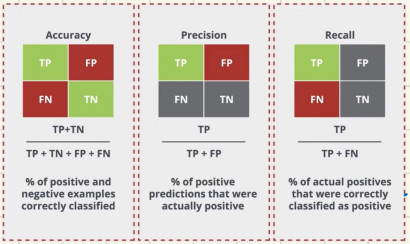
→ **CONFUSION MATRIX**

■ When the prediction matches the outcome we have either these result.

FALSE POSITIVE: predict positive label but actual value = negative

CONFUSION MATRIX:

- When building confusion matrix we would fill in the cells with accounts observed from our evaluation data set.
- We can then use these counts to compute different classification metrics.
- 3 most common classification metrics will derive from confusion matrix:



GREEN ELEMENTS: elements in the numerator

RED ELEMENTS: elements that are added to the GREEN to make the denominator

METRICS: Percentage or rate that captures different concepts.

- **ACCURACY** (AKA. ZERO-ONE LOSS): represents % of all classifications that match the actual value
  - [commonly used in multiclass problem]
- **PRECISION**: Focuses on the positive predictions and tells us what % of those were classified correctly.
  - [favored in binary classification when FP are worse than FN]
- **RECALL**: Focuses on the actual positive cases and tells us what % of the actual positives were classified as positives.
  - [favored in binary classification when FN are much worse than FP]

when using these metrics: between models

Practical Tips: always use a baseline for comparison

Base Rate = the percent of examples where Y==1

Problem 1	Problem 2
Base Rate = 10%	Base Rate = 50%
Accuracy = 90%	Accuracy = 70%

↑ NOT that useful of a model

IDEA:

BASE RATES: the % cases in your evaluation data where y=1 or it's average value of Y assuming its coded up as 1 and 0.

∴ When looking at specific measure of accuracy, precision, or recall, the value we put on that specific measure depends on what the base rate is.

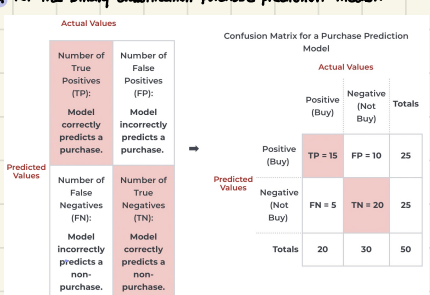
- As you iterate the baseline can be the most recent best performing model.
- In either case, the emphasis should be the relative improvement over the baseline, not necessarily the absolute number of the metric [APPLIES TO ALL 3 METRICS]

CLASSIFICATION EVALUATION METRICS

- Goal in training ML model: generalizes to new, unseen data.
- We need an objective way of measuring the performance of the model ∴ evaluation metrics.

Assume we built a binary classifier that predicts whether a customer will make a specific purchase, the model is predicting one of two class labels: "Buy"/"Not Buy".

Examine the confusion matrix for the binary classification purchase prediction model:



3 EVALUATION METRICS:

1. **ACCURACY**:
   - May overstate the fit of a model as we need to account the accuracy of guessing or by being correct by chance.
     - Our model "Buys" could have matched the actual "Buys" by chance w/ probability of 25/50 * 25/50 = 0.2
     - The "Not Buys" could have matched with 25/50 * 30/50 = 0.3
     - ∴ we could have an accuracy of 0.2 + 0.3 = 0.5
     - KAPPA: used to adjust our model accuracy by this random accuracy:

$$\text{Kappa} = \frac{(\text{Model Accuracy} - \text{Random Accuracy})}{(1 - \text{Random Accuracy})}$$

Ex) $\frac{0.7 - 0.5}{1 - 0.5} = 0.4$

$$\text{MODEL ACCURACY} = \frac{\#\text{ of correct predictions}}{\text{total }\#\text{ of predictions}} = \frac{(TP + TN)}{(TP + TN + FP + FN)}$$

Ex) # of correct predictions = 35 [15+20 on diagonal]
out of 50 total predictions:

$$\text{Model accuracy} = \frac{TP + TN}{TP + TN + FP + FN} = \frac{15 + 20}{15 + 20 + 10 + 5} = 0.70 = 70\%$$

② **PRECISION**: how well the model can predict a specific outcome; it's a measure of quality. [FOCUSES ON QUALITY]

$$\text{MODEL PRECISION} = \frac{\#\text{ of correct purchase predictions}}{\text{total \# of purchase predictions}} = \frac{TP}{TP+FP}$$

EX) Predicts 25 purchases (sum of 15+10)
Against those 15 of those purchase:

$$\text{Precision} = \frac{15}{15+10} = 0.60 = 60\%$$

③ **RECALL**: # of times the model correctly predicted a specific outcome. [FOCUSES ON QUATITY]
Calculates the proportion of actual positive classes that were predicted AKA. true positive rate.

$$\text{MODEL RECALL} = \frac{\#\text{ of correct purchase predictions}}{\text{total \# of actual purchase}} = \frac{TP}{TP+FN}$$

EX) 20 actual purchase
The model correctly predicts 15 of those purchase

$$\text{Recall} = \frac{15}{15+5} = 0.75 = 75\%$$

CHOOSING A CLASSIFICATION THRESHOLD

RECALL: output of logistic regression model is a probability that an example belongs to a given class.
To map the probabilities back to class label we have to choose a **THRESHOLD**.

Determining threshold can enable you to choose the appropriate threshold that results the best model performance.

RECALL:
- Low threshold ⟹ high recall value ⟹ capturing as many positive cases as possible
  High threshold ⟹ low recall value ⟹ capture few of the most positive cases.

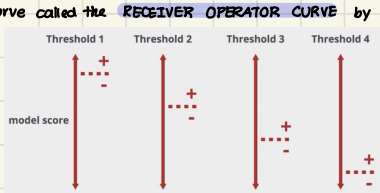
PRECISION:
- We need to reduce the # of FP by setting the threshold we'll ensure precision is maximized.
- To help us better understand the model's precision tradeoff for various values & determine optimal threshold to choose: **AUC-ROC TOOL**

EVALUATING A CLASSIFIER USING THE AUC-ROC CURVE

When you don't want to commit to a single threshold:
- You're uncertain about false negatives/positive costs
- You're uncertain about our prediction budget

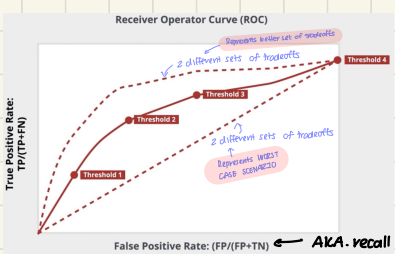
} When either of these conditions are met use the AUC as our default metric.

AREA UNDER THE RECEIVER OPERATOR CURVE (AUC): used for metric ranking.
- Constructing an AUC:
  - Build a curve called the **RECEIVER OPERATOR CURVE** by iterating through every possible threshold in the data:



Each vertical line is the same ranked list of model scores.

  - With each threshold we compute the FP rate and the TP rate from the confusion matrix and plot those points:

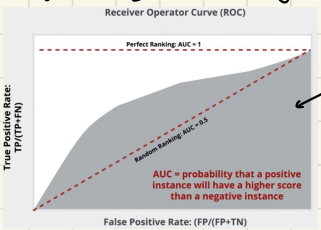


→ If we connect these points with the curve we end up with the **RECEIVER OPERATOR CURVE (ROC)**
- Represents the trade offs we'll make if we want to use our model for classification.
- Better models will always have an ROC that's steeper & higher on the left.

False Positive Rate: (FP/(FP+TN) ← AKA. recall ← To get a certain recall, we need to tolerate a certain level of false positives

WORST CASE SCENARIO: performs like a random # generator

- Capture the average in a single metric by taking the area under this particular curve:



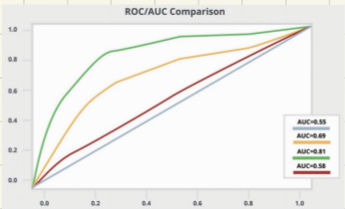
AUC is represented by the GRAY AREA

ROC would be a straight line b/c x & y axis have the range of 0 to 1.
The AUC for this random baseline is 0.5.

AUC = 1 is when it's a PERFECT MODEL meaning it ranks all positive instances above the negative instances so we can get perfect recall with no false positives.

↳ The AUC represents the probability that a randomly chosen positive example from your data will have a higher model score than a randomly chosen negative score from the data.

↳ AUC and ROC are useful tools for understanding the performance of binary classification models ∴ it's common to plot different model candidates against each other so that we can visualize the relative performances.



When doing automated model selection we can just compare AUC for each specific model candidate.

↳ REMEMBER: if you're unsure about the optimal threshold for classification threshold for your problem & you don't want to commit to a specific FP & TP rate then AUC is a safe metric.

METRICS FOR PROBABILITY ESTIMATION

NOTE: y is the variable we're trying to understand
x is the set of feature set to a particular value

EXPECTED VALUES (E[y]): What's the likely average future value of y.

CONDITIONAL EXPECTED VALUES (E[y|x]): what's the likely average future value of y in cases where X is true.

↳ When building a model we're using building a function aimed at computing the conditional expected value

↳ Metrics we use for this type of use cases are the same loss functions most commonly associated with training, classification, and regression models.

Classification $Log - Loss = (-1/N) * \sum_{i=1}^{N} y_i * log(p_i) + (1 - y_i) * log(1 - p_i)$ ← View this as metric we want to MINIMIZE. Use LOG LOSS formula to tune the model weights during training.

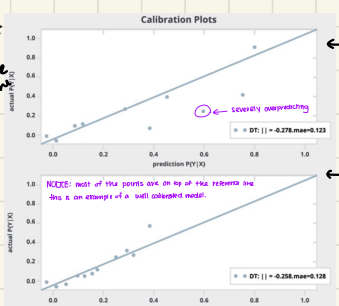
Regression $MSE = (1/N) * \sum_{i=1}^{N} (Y - f(X))^2$

- In classification case we use log loss for expected value measurement has to do with the idea of calibration.

MODEL CALIBRATION: If we estimate E[Y|x] as p, does Y actually happen at a rate of p in future examples. Used to measure if our probabilities are accurate.

↳ we can take the idea of calibrations to build curves that tells us exactly how well our model is producing accurate probabilities.

If we predict 20%, we want to see the average label be 20% at the limit



← Shows a calibration plot from a DT that's not trained on log loss but can still be evaluated using log loss.

Calibration plot
← Logistic regression which is trained on log loss

The calibration plot is built by grouping all of the examples by their predictions (range 0-1)

Then computing the average of the label for each of the prediction group.

- MEAN SQUARED ERROR (MSE) & LOG LOSS are metrics used for model selection and validation.
  - ⇓ Regression problems
  - ⇓ Classification problems

- Calibration plot is a tool used to visually verify whether our classification algorithm is producing accurate probabilities.
- Use calibration with other metrics to get a full sense of how well the model is performing.