DI501 Introduction to Data Informatics

Lecture 5: Model Configuration and Testing

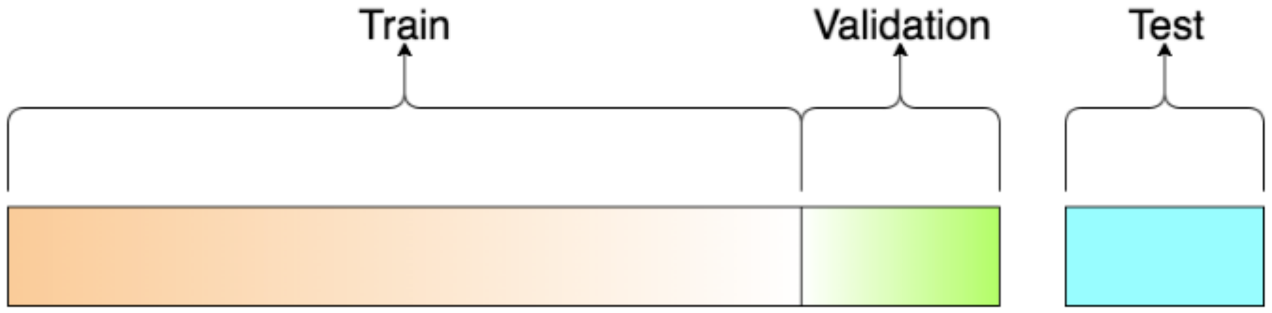
Classification- Part3

Training, Validation, Testing

***Training Dataset**:* The sample of data used to fit the model*.*

***Validation Dataset**:* The sample of data used to provide an unbiased evaluation of a model fit on the training dataset while tuning model hyperparameters. The evaluation becomes more biased as skill on the validation dataset is incorporated into the model configuration.

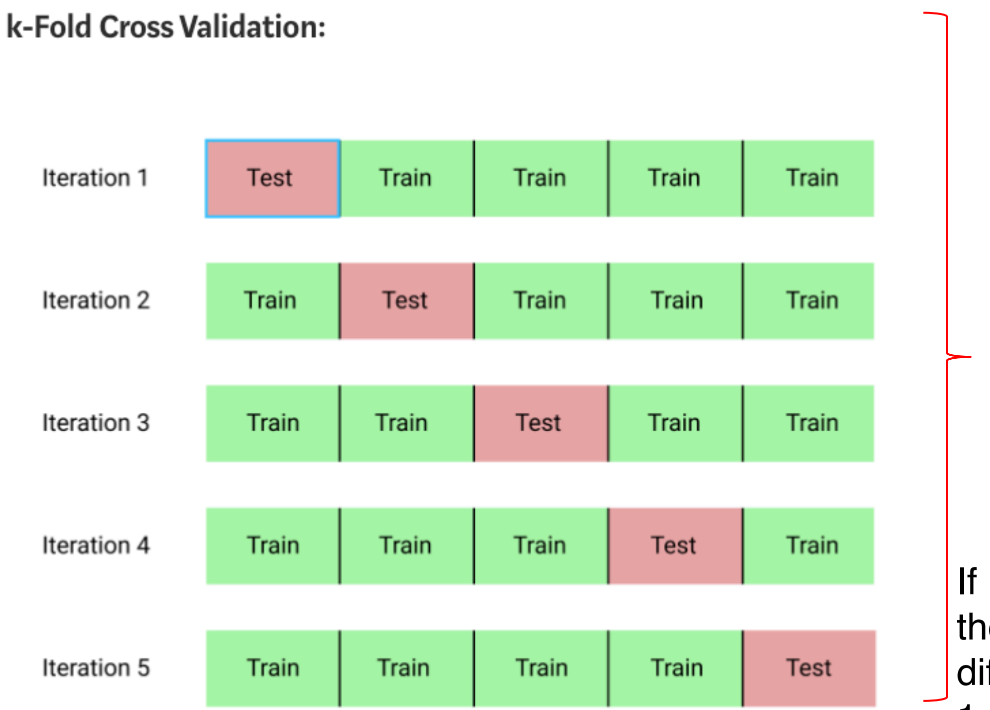
***Test Dataset**:* The sample of data used to provide an unbiased evaluation of a final model fit on the training dataset.



A visualisation of the splits

Ref: https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7

Types of Cross Validation



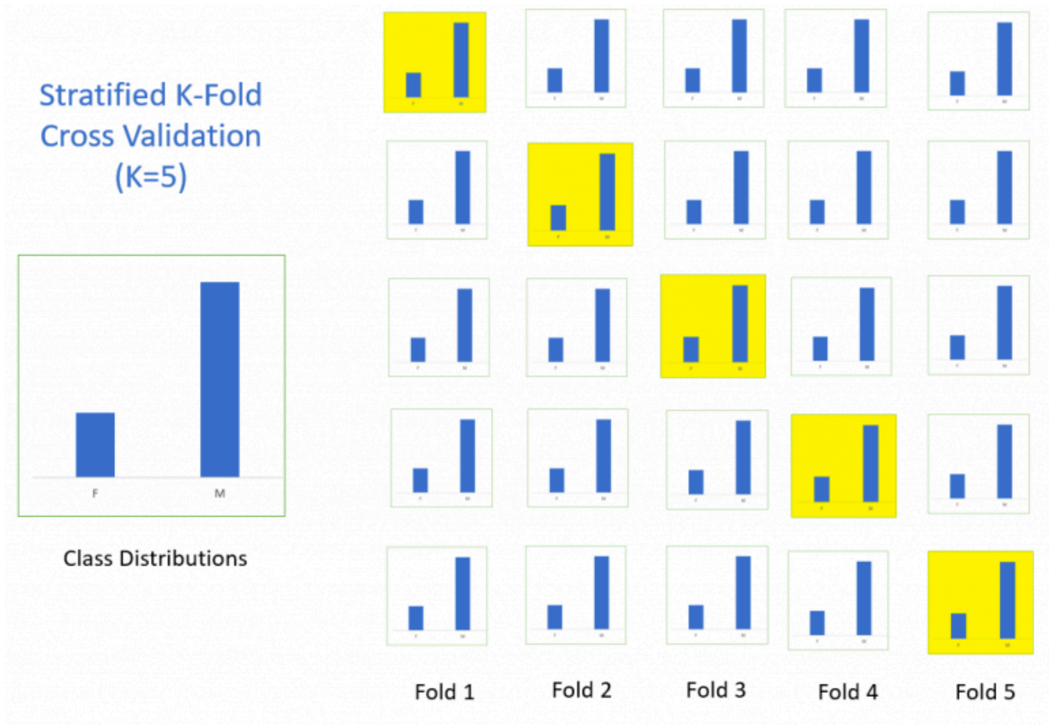
Get the average of the five models.

If k=5 the dataset will be divided into 5 equal parts and the below process will run 5 times, each time with a different holdout set.

1. Take the group as a holdout or test data set
2. Take the remaining three groups as a training data set
3. Take one random group as a validation dataset
4. Fit a model on the training set and find the best parameters using validation dataset
5. Evaluate it on the test set
6. Retain the evaluation score and discard the model

Ref: https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7

Types of Cross Validation

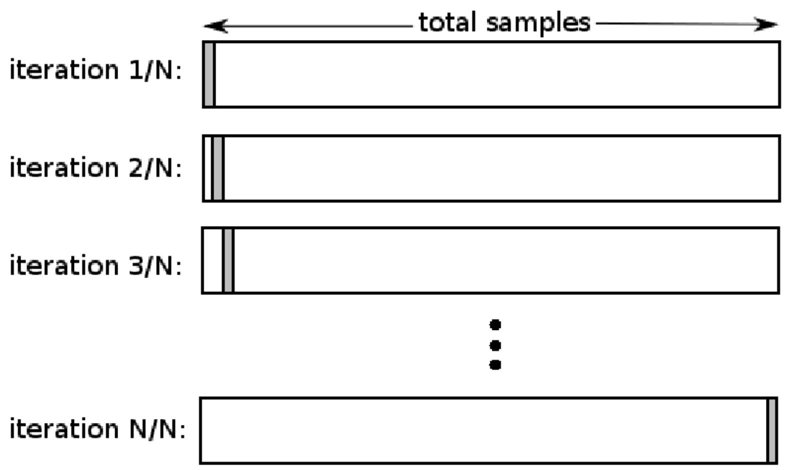


The splitting of data into folds may be governed by criteria such as ensuring that each fold has the same proportion of observations with a given categorical value, such as the class outcome value. **This is called stratified cross-validation.**

Ref: https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7
https://towardsdatascience.com/cross-validation-explained-evaluating-estimator-performance-e51e5430ff85

Types of Cross Validation

Leave one Out



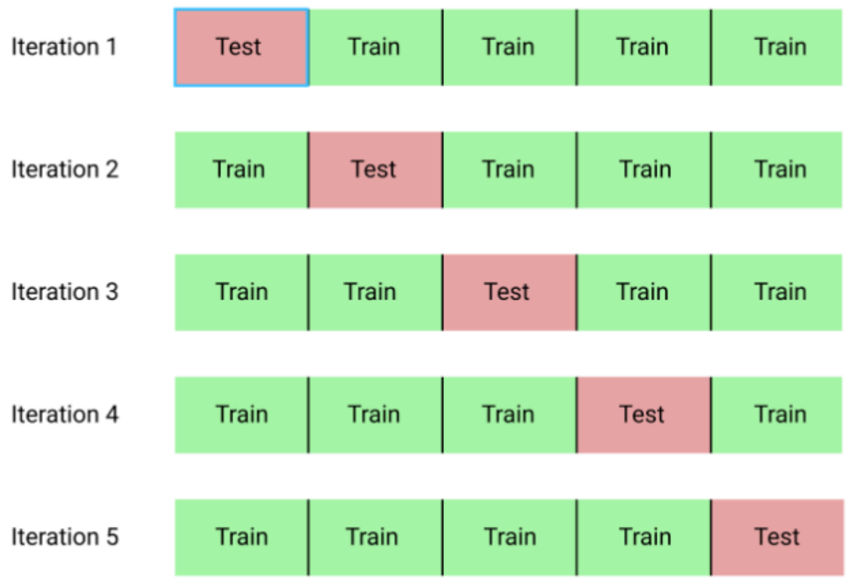
Leave One Out Cross Validation (LOOCV):
This approach leaves 1 data point out of training data, i.e. if there are n data points in the original sample then, n-1 samples are used to train the model and p points are used as the validation set.
This is repeated for all combinations in which the original sample can be separated this way, and then the error is averaged for all trials, to give overall effectiveness.
This is preferred when we are dealing with small datasets.

The accuracy of the model is the total error obtained using all dataset

Types of Cross Validation

Nested Cross Validation

k-Fold Cross Validation:

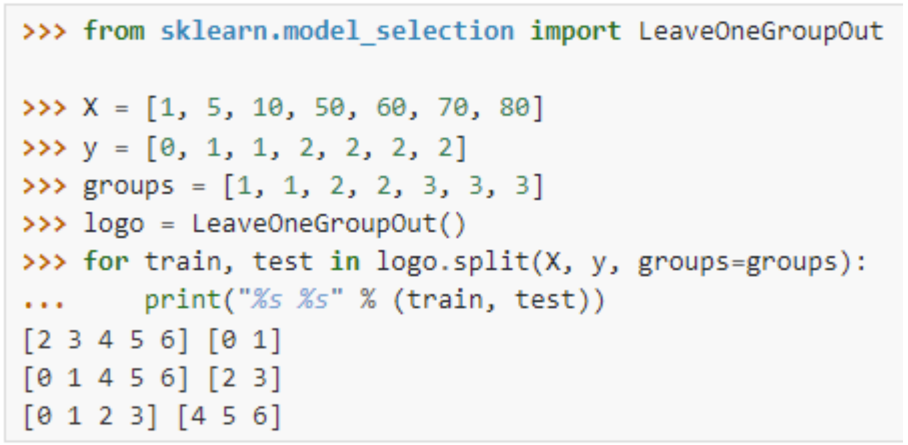


If k=5 the dataset will be divided into 5 equal parts and the below process will run 5 times, each time with a different holdout set.

```
1.   Take the group as a holdout or test data set
2.   Hold the four groups and assign it to "Group" vector
for (i=1: 4)
    Assign Group(i) to validation dataset
    Assign the remaining groups to training dataset
    Fit the model to the training dataset while finding the best parameters using Group(i) validation dataset
    Save the model to Model(i) and its performance to P(i)
end
3. Find the best model b among P
5. Evaluate it on the test set
6. Retain the evaluation score and discard the model
```

Leave One Group Out

- It is a cross-validation scheme which holds out the samples according to a third-party provided array of integer groups.
- This group information can be used to encode arbitrary domain specific pre-defined cross-validation folds.



```
>>> from sklearn.model_selection import LeaveOneGroupOut

>>> X = [1, 5, 10, 50, 60, 70, 80]
>>> y = [0, 1, 1, 2, 2, 2, 2]
>>> groups = [1, 1, 2, 2, 3, 3, 3]
>>> logo = LeaveOneGroupOut()
>>> for train, test in logo.split(X, y, groups=groups):
...     print("%s %s" % (train, test))
[2 3 4 5 6] [0 1]
[0 1 4 5 6] [2 3]
[0 1 2 3] [4 5 6]
```

Leave One Group Out

- We use it for showing model's generalization on unseen group items or individuals.
  - **Example**: There are *n* individuals and each watched 1-10 videos.
  - You built a classifier which is used to estimate each individual's engagement with a video.
    - Your model is trained with *n-1* individuals data.
    - You predict the model performance on *nth* individual videos (1-10 video). Here the group is the videos of *nth* individual.
    - You repeat this procedure for *n* individuals.
    - You report the overall performance.
    - If the number of individuals is small, you can report each individual's performance separately.

https://scikit-learn.org/stable/modules/cross_validation.html#leave-one-group-out

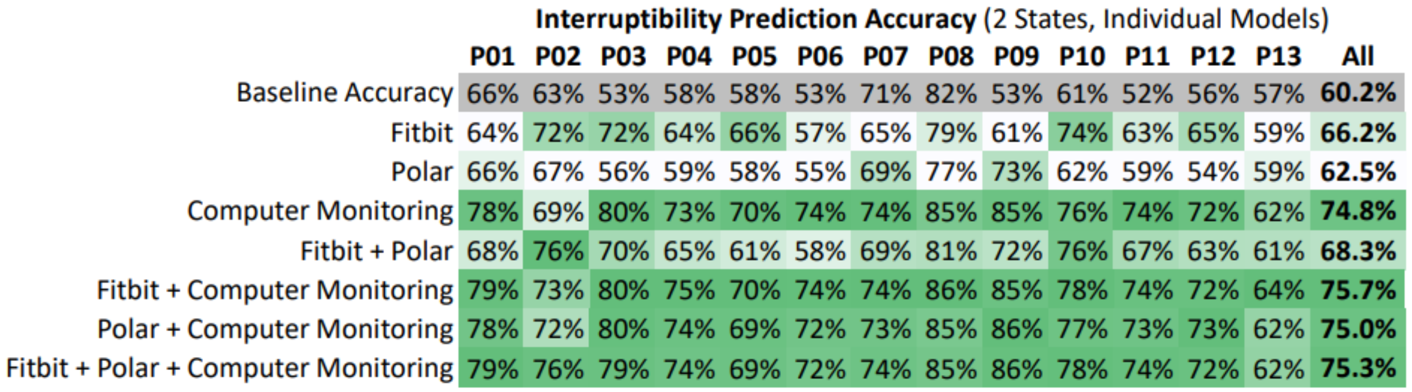
Leave one out individual (repeated measure)
Example: CHI 2018 Honourable Mention

- Motivation:
  - Knowledge workers experience many interruptions during their work day. Especially when they happen at inopportune moments, interruptions can incur high costs, cause time loss and frustration.
  - Knowing a person's interruptibility allows optimizing the timing of interruptions and minimize disruption.
  - This study aims to predict one’s interruptible moments.

Züger, M., Müller, S. C., Meyer, A. N., & Fritz, T. (2018, April). Sensing interruptibility in the office: A field study on the use of biometric and computer interaction sensors. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (pp. 1-14).

Leave one out individual (repeated measure)

Example: CHI 2018 Honourable Mention



Interruptibility Prediction Accuracy (2 States, Individual Models)

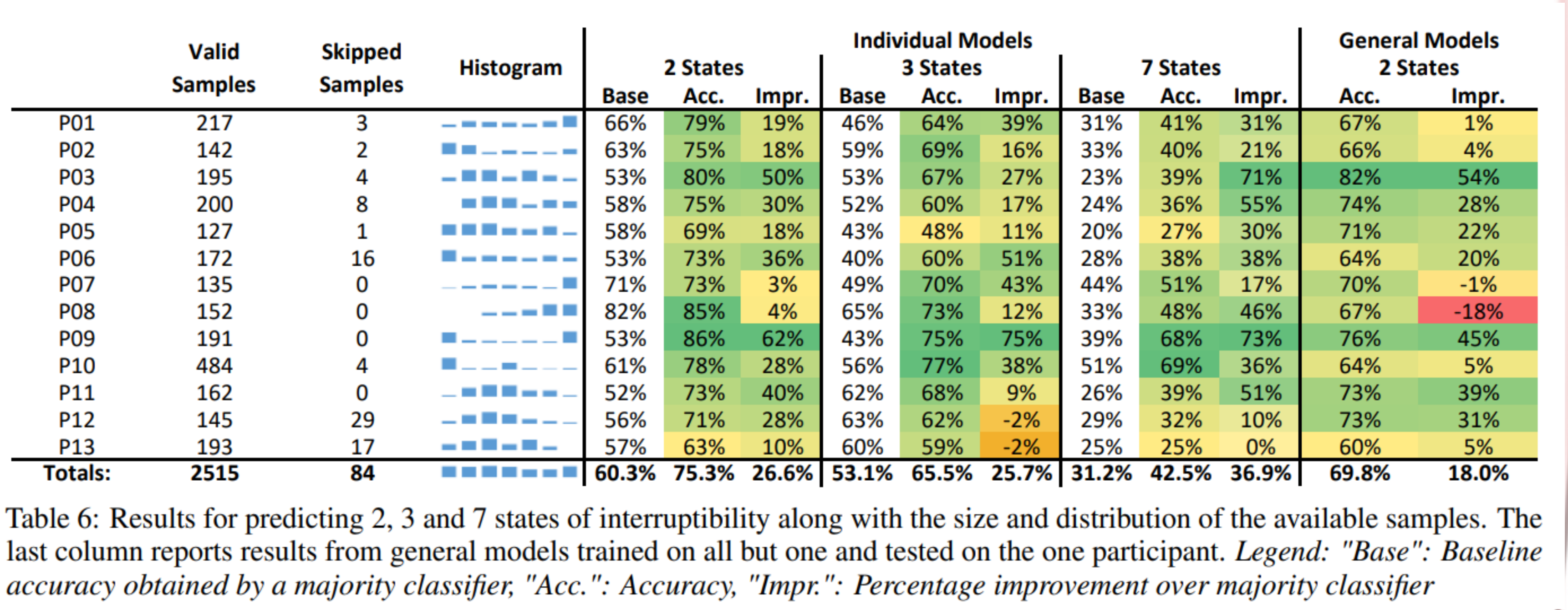
	P01	P02	P03	P04	P05	P06	P07	P08	P09	P10	P11	P12	P13	All
Baseline Accuracy	66%	63%	53%	58%	58%	53%	71%	82%	53%	61%	52%	56%	57%	**60.2%**
Fitbit	64%	72%	72%	64%	66%	57%	65%	79%	61%	74%	63%	65%	59%	**66.2%**
Polar	66%	67%	56%	59%	58%	55%	69%	77%	73%	62%	59%	54%	59%	**62.5%**
Computer Monitoring	78%	69%	80%	73%	70%	74%	74%	85%	85%	76%	74%	72%	62%	**74.8%**
Fitbit + Polar	68%	76%	70%	65%	61%	58%	69%	81%	72%	76%	67%	63%	61%	**68.3%**
Fitbit + Computer Monitoring	79%	73%	80%	75%	70%	74%	74%	86%	85%	78%	74%	72%	64%	**75.7%**
Polar + Computer Monitoring	78%	72%	80%	74%	69%	72%	73%	85%	86%	77%	73%	73%	62%	**75.0%**
Fitbit + Polar + Computer Monitoring	79%	76%	79%	74%	69%	72%	74%	85%	86%	78%	74%	72%	62%	**75.3%**

- ➢ Prediction results using different sensors and combinations thereof per participant (13 in total) and averaged over all (the darker the color the higher the accuracy).
- ➢ As baseline accuracy they report the accuracy that a majority classifier would achieve that always predicts the class containing more samples. The results are obtained training individual models for two states of interruptibility.
- ➢ While all sensors were better than the baseline, the features of the computer interaction sensors (accuracy=74.8%) were more predictive compared to the features from the biometric sensors (accuracy=68.3%).
- ➢ Adding one or both biometric sensors slightly improves the classifier (accuracy=75.7%).

Züger, M., Müller, S. C., Meyer, A. N., & Fritz, T. (2018, April). Sensing interruptibility in the office: A field study on the use of biometric and computer interaction sensors. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (pp. 1-14).

Leave one out individual (repeated measure)

Example: CHI 2018 Honourable Mention



	Valid Samples	Skipped Samples	Histogram	Individual Models 2 States Base	2 States Acc.	2 States Impr.	3 States Base	3 States Acc.	3 States Impr.	7 States Base	7 States Acc.	7 States Impr.	General Models 2 States Acc.	General Models 2 States Impr.
P01	217	3		66%	79%	19%	46%	64%	39%	31%	41%	31%	67%	1%
P02	142	2		63%	75%	18%	59%	69%	16%	33%	40%	21%	66%	4%
P03	195	4		53%	80%	50%	53%	67%	27%	23%	39%	71%	82%	54%
P04	200	8		58%	75%	30%	52%	60%	17%	24%	36%	55%	74%	28%
P05	127	1		58%	69%	18%	43%	48%	11%	20%	27%	30%	71%	22%
P06	172	16		53%	73%	36%	40%	60%	51%	28%	38%	38%	64%	20%
P07	135	0		71%	73%	3%	49%	70%	43%	44%	51%	17%	70%	-1%
P08	152	0		82%	85%	4%	65%	73%	12%	33%	48%	46%	67%	-18%
P09	191	0		53%	86%	62%	43%	75%	75%	39%	68%	73%	76%	45%
P10	484	4		61%	78%	28%	56%	77%	38%	51%	69%	36%	64%	5%
P11	162	0		52%	73%	40%	62%	68%	9%	26%	39%	51%	73%	39%
P12	145	29		56%	71%	28%	63%	62%	-2%	29%	32%	10%	73%	31%
P13	193	17		57%	63%	10%	60%	59%	-2%	25%	25%	0%	60%	5%
Totals:	**2515**	**84**		**60.3%**	**75.3%**	**26.6%**	**53.1%**	**65.5%**	**25.7%**	**31.2%**	**42.5%**	**36.9%**	**69.8%**	**18.0%**

Table 6: Results for predicting 2, 3 and 7 states of interruptibility along with the size and distribution of the available samples. The last column reports results from general models trained on all but one and tested on the one participant. *Legend: "Base": Baseline accuracy obtained by a majority classifier, "Acc.": Accuracy, "Impr.": Percentage improvement over majority classifier*

Important Notes:

1. Baseline is important.
2. If your study is repeated measure study, it might be good to report on the subject individually.
   Summarization scores such as accuracy will be biased towards the common patterns.

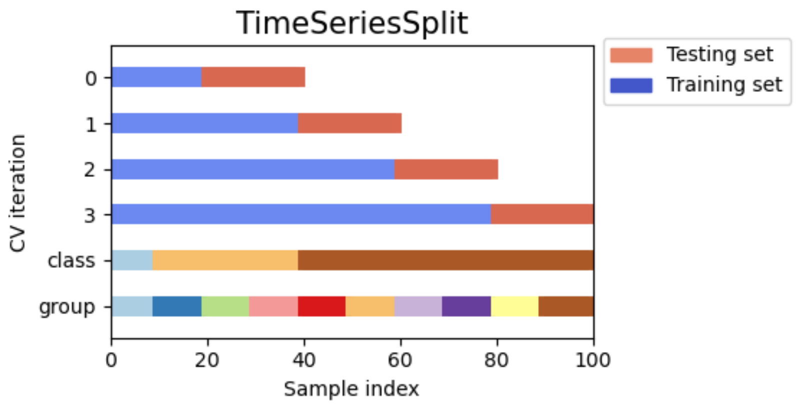
Züger, M., Müller, S. C., Meyer, A. N., & Fritz, T. (2018, April). Sensing interruptibility in the office: A field study on the use of biometric and computer interaction sensors. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (pp. 1-14).

Timewise Split (Out of time sampling)

- Some domains have time dimensions.
  - Example: Social media posts
- Dataset is sorted according to time.
  - It is split according to a time point.
  - The first split is training, the second split is validation and the last one is reserved as testing dataset.
- It is important how you choose the split time points.
  - It needs to cover sufficient information with respect to characteristics of the dataset.
  - Example: cycles, seasonalities, etc.

Timewise Split (Out of time sampling)

- There is still a possibility of leakage.
  - Especially if you reserve a large amount of data for training, there is still a chance to see similar instances (such as posts) in the remaining test dataset.
  - Therefore, as a complementary approach, to discuss about generalization properly, you can try incremental time splits:
    - Example: %30,%50,%70,%90 time splits



Model Tuning

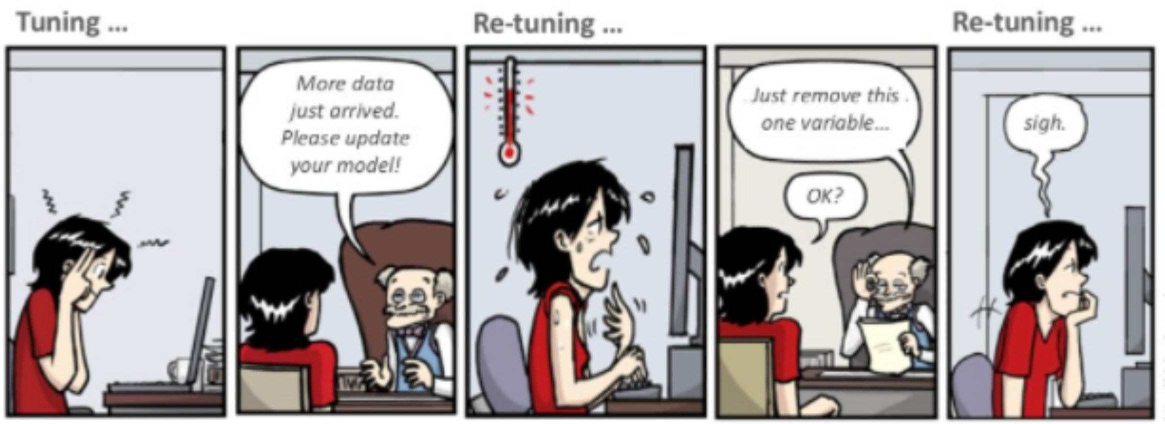
Model Parameter Setting

- Understand the model input parameters thoroughly when you apply a model.
  - Which ones are important for overfitting?
  - Do some of them depend on your dataset characteristics?
- Justify your parameters wisely.
  - You should not state that you have already used the default parameters.
    - It is not a justification!
- Hyperparameters are values which are set before the learning process.
  - You need to tune them as models do not learn these.
  - They affect the performance significantly.

Model Tuning

Model Parameter Search

- You can look at similar domain papers and tutorials to understand which hyperparameters are very important.
  - Choose the resources from trusted/peer reviewed journals and conferences.
    - Because it may be overlooked in numerous papers.
    - You can probably find at least one paper which does not do this justification!
      - Therefore choosing the right paper for referencing is important.
- How to automate hyperparameter search?
  - Grid Search
  - Random Search
  - Informed Search


Tuning ...
More data just arrived. Please update your model!
Re-tuning ...
Just remove this one variable...
OK?
Re-tuning ...
sigh.

Model Tuning

Model Parameter Search: Grid Search

- Select values for each parameters to set namely a search space grid.
- Try all combinations
- Example:
  - You try different k values for kNN algorithm.
  - You select the best performing one in the validation dataset.

```python
neighbors_list = [3,5,10,20,50,75]
accuracy_list = []
for test_number in neighbors_list:
    model = KNeighborsClassifier(n_neighbors=test_number)
    predictions = model.fit(X_train, y_train).predict(X_test)
    accuracy = accuracy_score(y_test, predictions)
    accuracy_list.append(accuracy)
```

Model Tuning

Model Parameter Search: Grid Search Example

- GridSearchCV class from Scikit-Learn library can be used for this optimization.
- Some of the most common hyperparameters are:
  - *n_neighbors*,
  - *weights* which can be set to either ‘uniform’, where each neighbor within the boundary carries the same weight or ‘distance’ where closer points will be more heavily weighted toward the decision.
    - Note that when weights = 'distance' the class with the highest number in the boundary may not “win the vote”
  - *metric* which refers to how the distance of neighboring points is chosen from the unknown point.

https://medium.com/@erikgreenj/k-neighbors-classifier-with-gridsearchcv-basics-3c445ddeb657

Model Tuning

Model Parameter Search: Grid Search Example

```
from sklearn.model_selection import GridSearchCV

grid_params = {
    'n_neighbors': [3,5,11,19],
    'weights': ['uniform', 'distance'],
    'metric':['euclidean', 'manhattan']
}

gs = GridSearchCV(
     KNeighborsClassifier(),
     grid_params,
     verbose = 1,
     cv = 3,
     n_jobs = -1
     )

gs_results = gs.fit(X_train, y_train)
```

When you do a GridSearch, you're running many more models than when you simply fit and score.
In the left GridSearch there are:
4 possibilities for `n_neighbors` *
2 possibilities for `weights` *
2 possibilities for `metric` *
3 cross-validations

For a total of 4 * 2 * 2 * 3 = 48 total times running the model.

It's important to set verbose so you'll get feedback on the model and know how long it may take to finish.

https://medium.com/@erikgreenj/k-neighbors-classifier-with-gridsearchcv-basics-3c445ddeb657

Model Tuning

Model Parameter Search: Grid Search Example

```
from sklearn.model_selection import GridSearchCV

grid_params = {
    'n_neighbors': [3,5,11,19],
    'weights': ['uniform', 'distance'],
    'metric':['euclidean', 'manhattan']
}

gs = GridSearchCV(
     KNeighborsClassifier(),
     grid_params,
     verbose = 1,
     cv = 3,
     n_jobs = -1
     )

gs_results = gs.fit(X_train, y_train)
```

When you do a GridSearch, you're running many more models than when you simply fit and score.
In the left GridSearch there are:
4 possibilities for `n_neighbors` *
2 possibilities for `weights` *
2 possibilities for `metric` *
3 cross-validations

For a total of 4 * 2 * 2 * 3 = 48 total times running the model.

It's important to set verbose so you'll get feedback on the model and know how long it may take to finish.

Another thing to notice from the GridSearch is that `n_jobs = -1`. By setting n_jobs to -1, you're telling the computer to use **all** of it's processors to perform the model.

Model Tuning

Model Parameter Search: Grid Search Example

To return the best parameters and score for your model from the grid search, use the following commands.

```
gs_results.best_score_
```

```
gs_results.best_estimator_
```

```
gs_results.best_params_
```

You can reserve your testing dataset at the beginning. You can apply *n*-fold cross-validation dataset and conduct this operation for *n* times.
Then, it will be called nested cross validation.
Finally, you will have *n* number of best models.

Disadvantages:
Computationally expensive.
It is “uninformed”. Results of one model does not help creating the next model.

https://medium.com/@erikgreenj/k-neighbors-classifier-with-gridsearchcv-basics-3c445ddeb657

Model Tuning

Model Parameter Search: Random Search

- The study of Bergstra and Bengio (*) shows empirically and theoretically that randomly chosen trials are more efficient for hyper-parameter optimization than trials on a grid.
- Motivation: Not all the features are important.

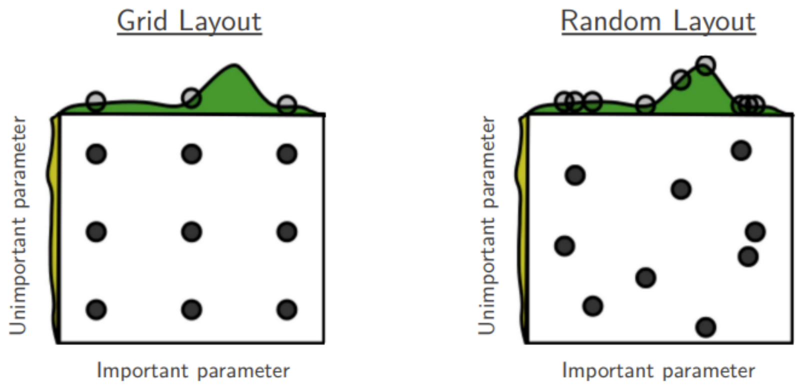
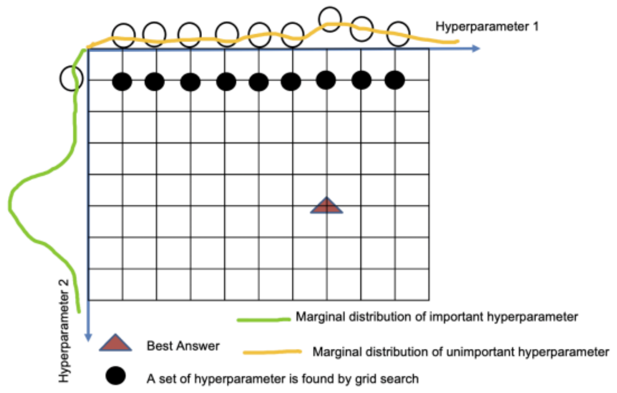


Figure 1: Grid and random search of nine trials for optimizing a function $f(x,y) = g(x) + h(y) \approx g(x)$ with low effective dimensionality. Above each square $g(x)$ is shown in green, and left of each square $h(y)$ is shown in yellow. With grid search, nine trials only test $g(x)$ in three distinct places. With random search, all nine trials explore distinct values of g. This failure of grid search is the rule rather than the exception in high dimensional hyper-parameter optimization.

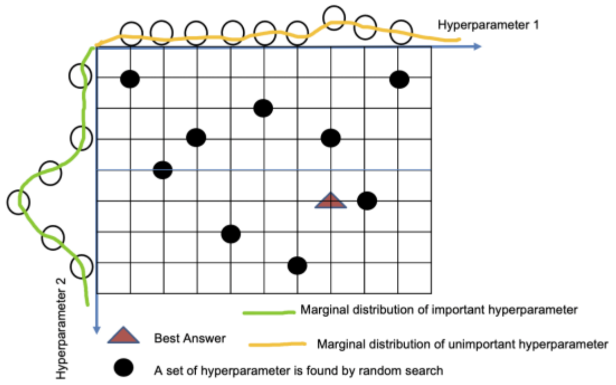
(*) Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of machine learning research*, *13*(2).

Model Tuning

Model Parameter Search: Random Search


Hyperparameter 1
Hyperparameter 2
Marginal distribution of important hyperparameter
Best Answer
Marginal distribution of unimportant hyperparameter
A set of hyperparameter is found by grid search

(a) Grid search


Hyperparameter 1
Hyperparameter 2
Marginal distribution of important hyperparameter
Best Answer
Marginal distribution of unimportant hyperparameter
A set of hyperparameter is found by random search

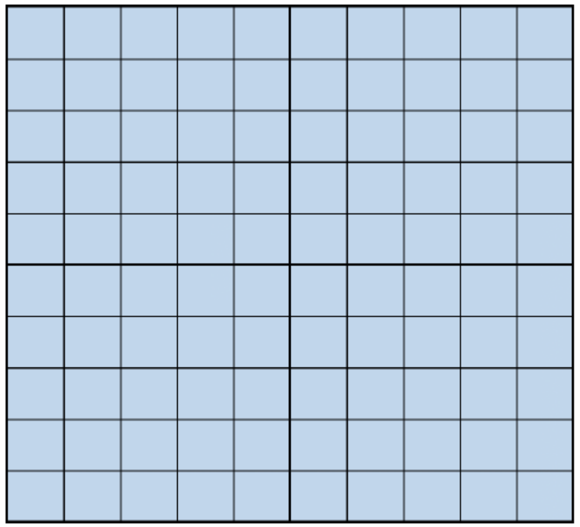
(b) Random search

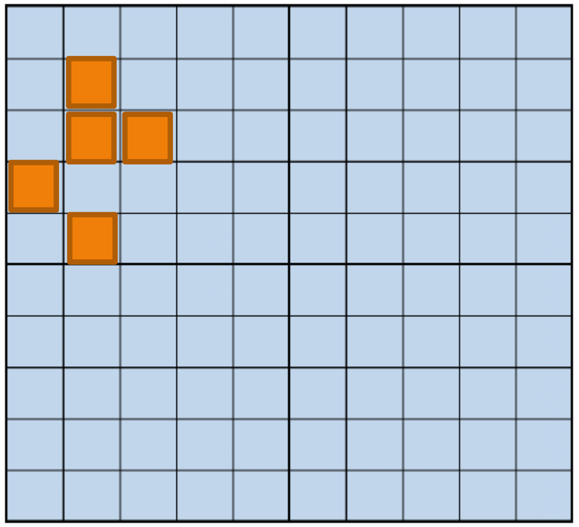
There are 10x10=100 parameters to try to find the best optimal one.
In grid search, the cost is 100.

With random search, we cover a lot of space.
Hence, with 10 searches, we approximate to the best parameter better compared to grid search.

Model Tuning

Model Parameter Search: Random Search



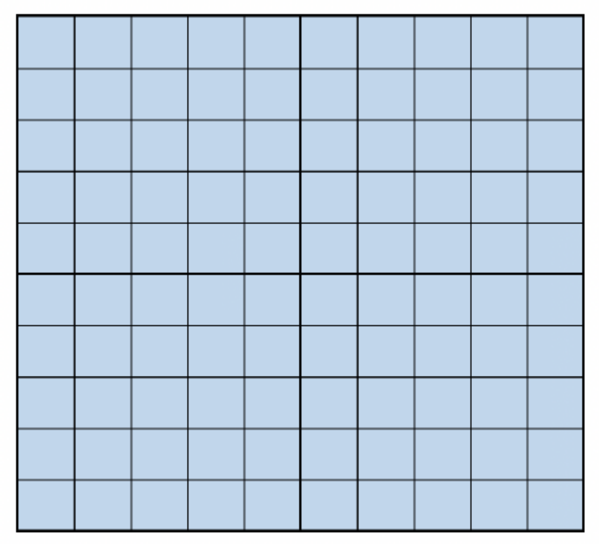


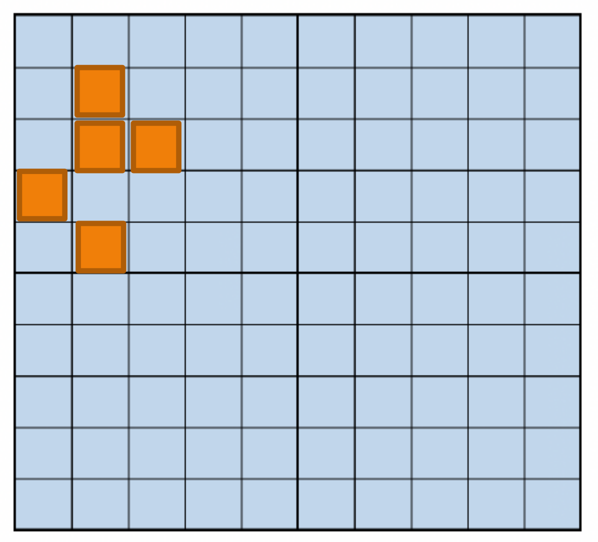
- 10x10=100 parameters to try.
- Assume that the best models are located in the orange boxes.
- How many models should I run to find one of the best models?
- 95 models with grid search!

(*) Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of machine learning research*, *13*(2).

Model Tuning

Model Parameter Search: Random Search





We need 59 models instead of 95 models!

- Let's select a point uniformly from the search space.

Run1	0.05 success	(1-0.05)=0.95 failure
Run2	(1-0.9)=0.1 success	0.95^2=0.90 failure
Run3	(1-0.85)=0.15 success	0.95^3=0.85 failure
Run*n*	(1-0.05)=0.95 success	0.95^n=0.05 failure (*n*=59)

(*) Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of machine learning research*, *13*(2).

Model Tuning

Model Parameter Search: Random Search

Example:

```
from sklearn.model_selection import RandomizedSearchCV
# specify "parameter distributions" rather than a "parameter grid“
k_range = list(range(1, 31))
weight_options = ['uniform', 'distance']
param_dist = dict(n_neighbors=k_range, weights=weight_options)
# n_iter controls the number of searches
rand = RandomizedSearchCV(knn, param_dist, cv=10, scoring='accuracy', n_iter=10, random_state=5, return_train_score=False)
rand.fit(X, y)
pd.DataFrame(rand.cv_results_)[['mean_test_score', 'std_test_score', 'params']]

#DataFrame

mean_test_score std_test_score  params
0   0.973333    0.032660    {'weights': 'distance', 'n_neighbors': 16}
1   0.966667    0.033333    {'weights': 'uniform', 'n_neighbors': 22}
2   0.980000    0.030551    {'weights': 'uniform', 'n_neighbors': 18}
3   0.966667    0.044721    {'weights': 'uniform', 'n_neighbors': 27}
4   0.953333    0.042687    {'weights': 'uniform', 'n_neighbors': 29}
5   0.973333    0.032660    {'weights': 'distance', 'n_neighbors': 10}
6   0.966667    0.044721    {'weights': 'distance', 'n_neighbors': 22}
7   0.973333    0.044222    {'weights': 'uniform', 'n_neighbors': 14}
8   0.973333    0.044222    {'weights': 'distance', 'n_neighbors': 12}
9.    0.973333    0.032660    {'weights': 'uniform', 'n_neighbors': 15}
# examine the best model
print(rand.best_score_)
print(rand.best_params_)

# Output
0.9800000000000001
{'weights': 'uniform', 'n_neighbors': 18}
```

https://dev.to/balapriya/hyperparameter-tuning-understanding-randomized-search-343l

Model Tuning

Model Parameter Search: Random Search

Parameters of the best model

- Surprisingly, we see that the highest accuracy score obtained in this case, where we only looked at 10 different parameter settings instead of 60 in Grid Search, is the same as before: 0.98 ✔
- And the value for n_neighbors= 18, which is also one of the optimal values that we got when we initially searched for the optimal value of n_neighbors.
- Even though Randomized Search may not always give the hyperparameters of the best performing model, the models obtained by using these hyperparameters do not perform much worse compared to the best model obtained from Grid Search.
    - This means, the best models thus obtained, with the hyperparameters from randomized search are clearly very close to the optimal model.

https://dev.to/balapriya/hyperparameter-tuning-understanding-randomized-search-343l

Model Tuning

Model Parameter Search: Informed Search

- Previous techniques are based on uninformed search, where hyperparameter learning doesn’t learn from previous iterations.
- Informed search techniques do the opposite.
- A basic informed search method:
  - Random search
  - Find the promising areas
  - Perform a grid search on those areas
  - Continue until optimal score is obtained

Model Tuning

Model Parameter Search: Informed Search

- Bayesian optimization:
  - In previous basic informed search, we started with equal probabilities for each instance in the search space.
    - Then we updated our beliefs once we found a promising area, meaning that the parameters in this area had a higher chance.
    - In other words, we updated our prior.

Model Tuning

Model Parameter Search: Informed Search

- Bayesian optimization starts by sampling randomly, e.g. 30 combinations, and computes the cross-validation metric for each of the 30 randomly sampled combinations using *k-fold cross-validation*.
- Then the algorithm updates the distribution it samples from, so that it is more likely to sample combinations similar to the good metrics, and less likely to sample combinations similar to the poor metrics.
- As it continues to sample, it continues to update the search distribution it samples from, based on the metrics it finds.

https://druce.ai/2020/10/hyperparameter-tuning-with-xgboost-ray-tune-hyperopt-and-optuna

Model Tuning

Model Parameter Search: Informed Search

- If good metrics are not uniformly distributed, but found close to one another in a Gaussian distribution or any distribution which we can model, then Bayesian optimization can exploit the underlying pattern, and is likely to be more efficient than grid search or naive random search.
- Solutions:
  - HyperOpt, Optuna

https://druce.ai/2020/10/hyperparameter-tuning-with-xgboost-ray-tune-hyperopt-and-optuna