# Hyperparameter Tuning with Python

Mentor: Pararawendy Indarjo

Hey I'm,

# **Pararawendy Indarjo**

I am a,

- CURRENTLY | **Senior DS at Bukalapak**
- 19 - 20 | **Data Analyst at Eureka.ai**

BSc Mathematics

MSc Mathematics

# Outline

- What are hyperparameters?
  - Why optimizing them?
- Cross Validation
- Tuning penalized regression
- Tuning K-nearest neighbors
- Tuning K-means clustering
- Tuning Random Forest

# What are hyperparameters?

- Hyperparameters are parameters whose values are **specified by the modeller in advance** (before training the model)

- In contrast: usual model parameter values are derived (learned) via model training

- Sample of hyperparameters:
  - Lambda (regularization parameter) in Ridge/LASSO
  - K (number of clusters) in K-Means algorithm, and K-Nearest Neighbors
  - Number of ensembled trees in Random Forest model
  - Etc

# Why Optimizing Hyperparameters?

01

## To Avoid Overfitting

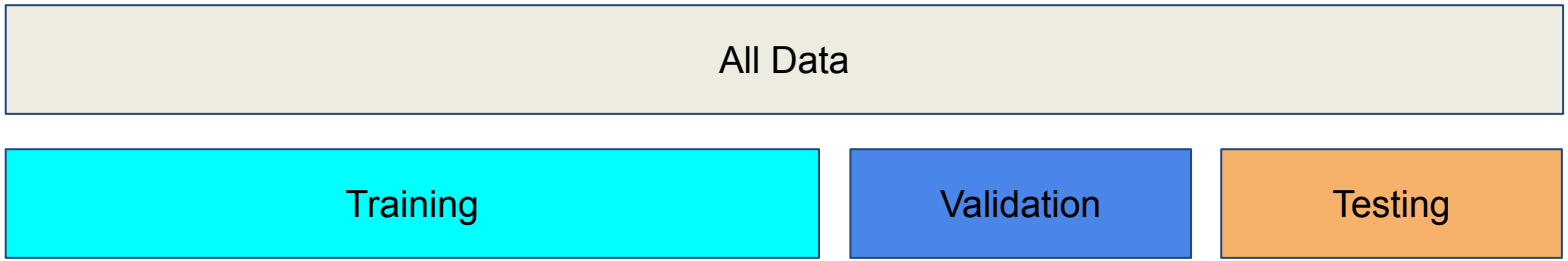
- The most complex model is **not** always better
- Our ultimate goal is to have a model that performs well on unseen data (test data)

02

## To Get Best of The Best Model

- Good data scientist NEVER build only a single model
- Build multiple models instead!
  - And choose the best one

# Recall: Our Strategy to Train Penalized Regression




**Train multiple models with different lambdas**

- Model 1 (lambda = 0.1)
- Model 2 (lambda = 1)
- Model 3 (lambda = 10)

**Choose the best lambda**

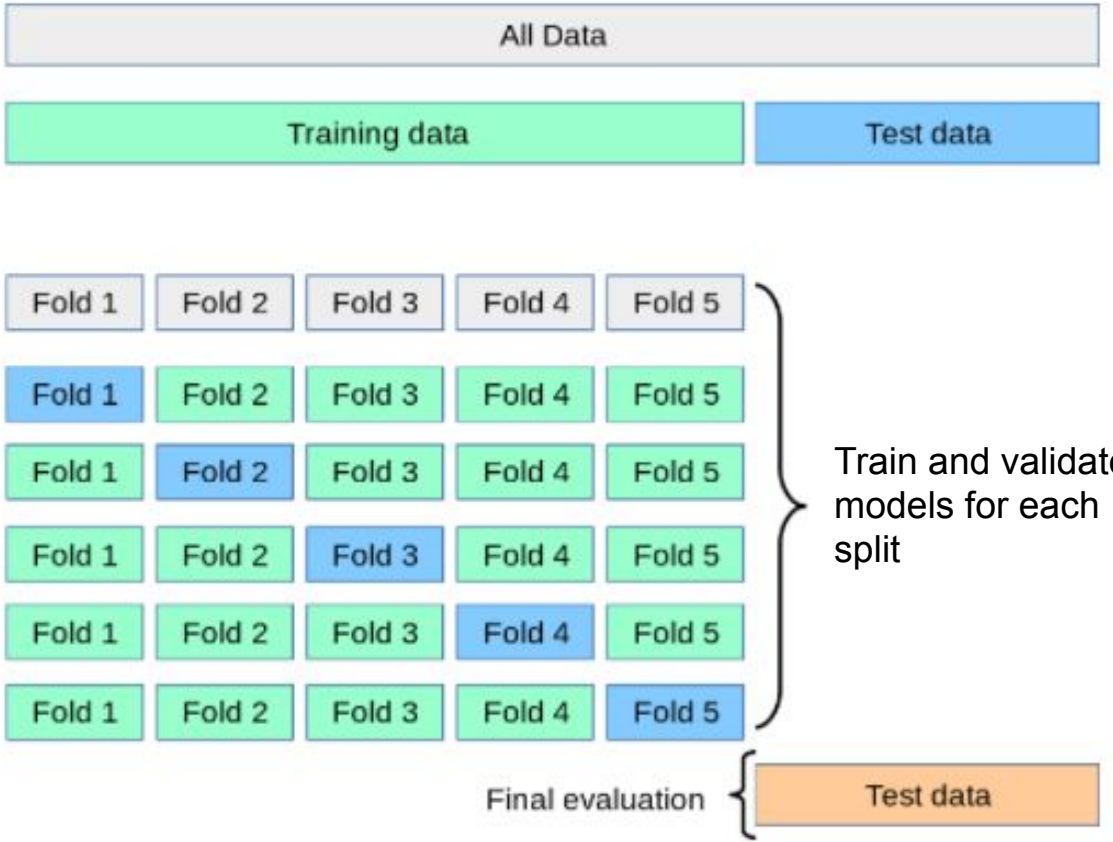
Lambda = 1 is best (Model 2)

**Evaluate the best model**

Report metrics of Model 2

# Cross Validation

Only divide full data into **two sets**




For each split, we leave 1 fold as **validation set**

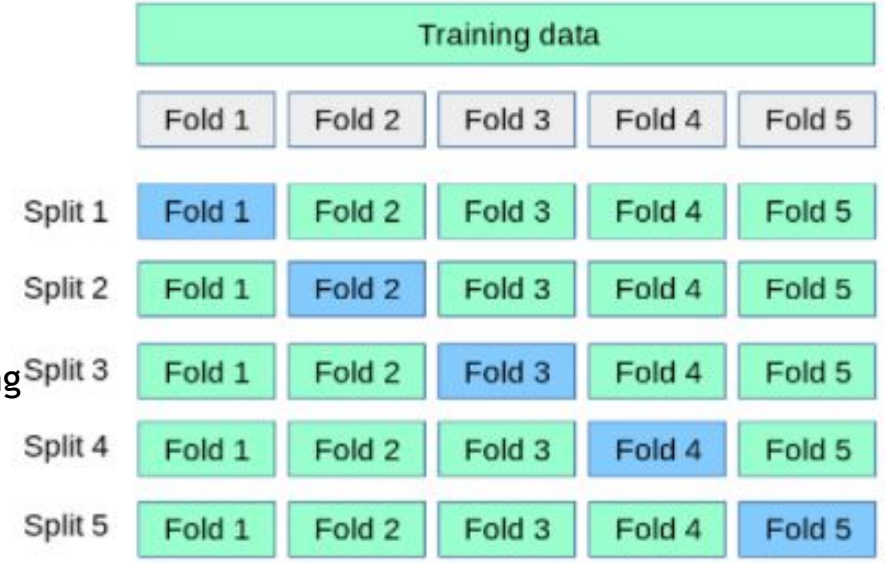
# K-Fold Cross Validation

- We separate the training data to become K parts (folds) with the same size

- For each split, we train the model on all K-1 folds (green folds)
  - And validate the model on the last fold that remains (blue fold)
  - i.e. compute evaluation metrics on the blue fold (can be seen as validation data)

- After done with all splits (combination of K-1 training parts and 1 validation part)
  - **Average out** all of K evaluation metrics to become the final metric
  - As basis to choose the best hyperparameter

- What number is K?
  - 5 is a good default
  - 3 is minimum, 10 when your machine allows



# +/- of Cross Validation

\+

## More robust results

- By validating the trained model multiple times, we will get more stable results
- And hence the correctness of the optimal hyperparameters chosen

\+

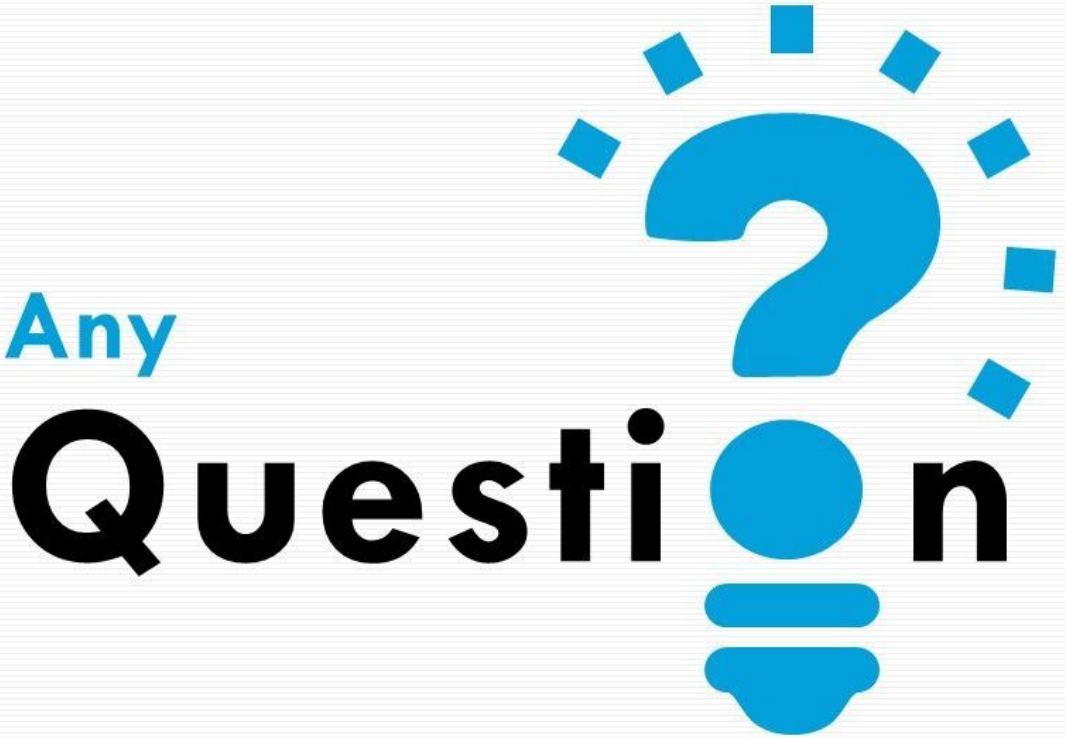
## Help maximizing data size value

- Cross validation is extremely helpful when our dataset is not too big
  - i.e. < 100K rows
- In a way, it gives us a free validation data

\-

## Heavier computation

- Since it takes K times training processes to finish one complete round of model training with a specific hyperparameter value


Any
Question

# Penalized Regression

We will tune **lambda**, which regulates the level of regularization of the regression

01

## Ridge

Lambda effect to the model coefficients

- Coeff will become smaller (larger lambda == smaller coeff)
- **Yet they all never be exactly zero**

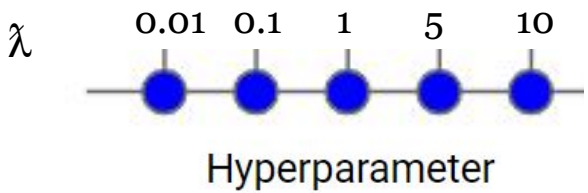
02

## LASSO

Lambda effect to the model coefficients

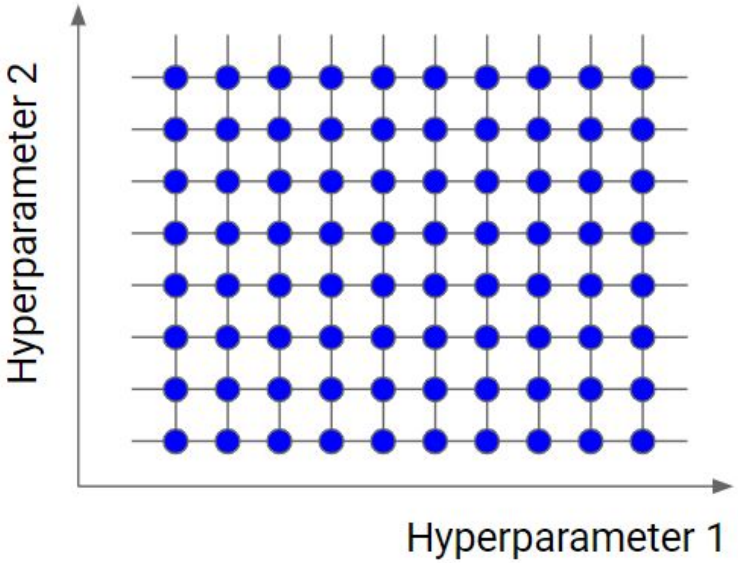
- Coeff will become smaller (larger lambda == smaller coeff)
- **Some of them will be exactly zero (eliminated from the model)**

# We'll use GridSearchCV function




- GridSearchCV is a function from **sklearn library** to do hyperparameter tuning based on Cross Validation
- Essentially, what it does is to **try every combination of the specified hyperparameter values**
- Example on the left:
  - There will be 5 times the program will do K-fold cross validation

# If > 1 types of hyperparameter




- GridSearchCV will **try every combination pair of the specified hyperparameter values**
- Example on the left:
  - Hyperparameter 1: 10 distinct values
  - Hyperparameter 2: 8 distinct values
  - So there are 10 x 8 = 80 times K-fold cross validation!

# Tuning Penalized Regression

For Ridge (LASSO is quite similar)

```
from sklearn.linear_model import Ridge
from sklearn.model_selection import GridSearchCV

ridge_reg = Ridge(random_state=42)
parameters = {
    'alpha': (0.000001,0.00001,0.0001,0.001,
              0.01, 0.1, 1, 5, 10, 20)
}

ridge_reg_gridcv = GridSearchCV(ridge_reg, parameters, cv=5,
                                scoring='neg_root_mean_squared_error')
ridge_reg_gridcv.fit(X_train, y_train)
```

*ensure reproducibility*

*tested lambda values*

*5-fold CV*

*Evaluation metric used*

# Tuning Penalized Regression

The results

```
retain_cols = ['params','mean_test_score','rank_test_score']
cv_result = pd.DataFrame(ridge_reg_gridcv.cv_results_)
cv_result[retain_cols]
```

| | params | mean_test_score | rank_test_score |
|---|---|---|---|
| 0 | {'alpha': 1e-06} | -55.972086 | 7 |
| 1 | {'alpha': 1e-05} | -55.972086 | 6 |
| 2 | {'alpha': 0.0001} | -55.972083 | 5 |
| 3 | {'alpha': 0.001} | -55.972053 | 4 |
| 4 | {'alpha': 0.01} | -55.971757 | 3 |
| 5 | {'alpha': 0.1} | -55.969020 | 2 |
| 6 | {'alpha': 1} | -55.959424 | 1 |

→ *Best model*

# Tuning Penalized Regression

The best model obtained

```
# best model
ridge_reg_gridcv.best_estimator_
# the coefficients of the best estimator (exclude intercept)
ridge_reg_gridcv.best_estimator_.coef_
```

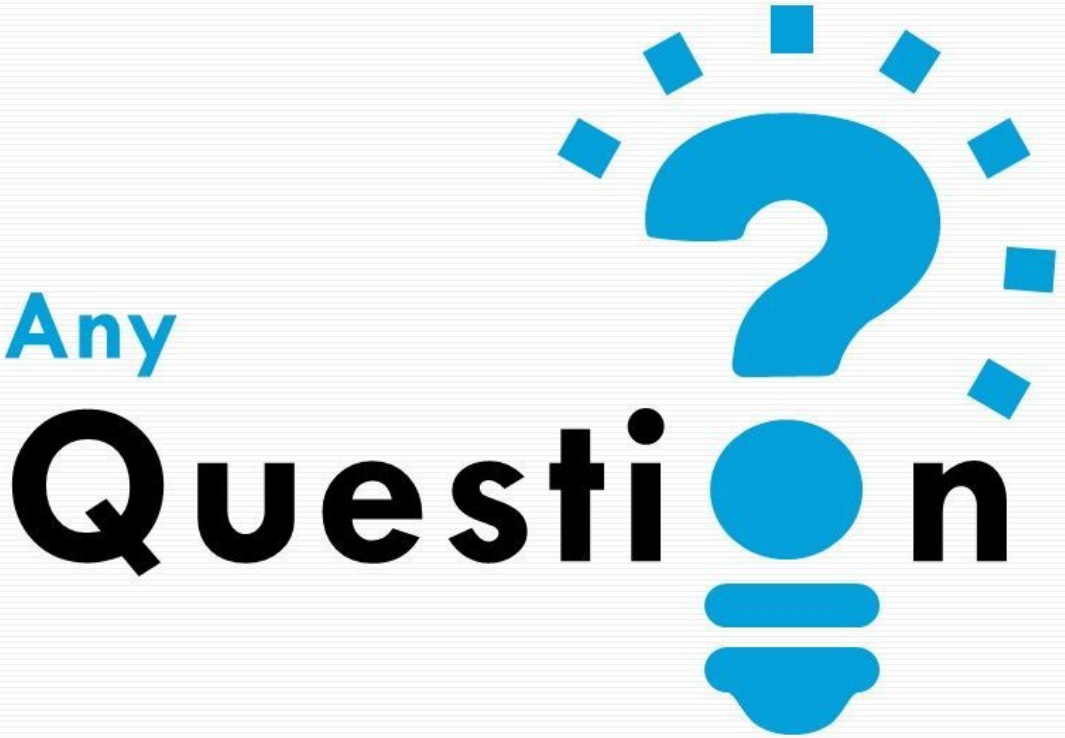
```
array([  0.14269511, -22.80861461,   5.90541998,   1.19867986,
        -1.07900835,   0.62662466,   0.3774738 ,   9.77013169,
        60.79394666,   0.21396887])
```

```
# the intercept of the best estimator
ridge_reg_gridcv.best_estimator_.intercept_
```

```
-319.81247103842134
```

# Hands-On

- Open today’s Jupyter notebook on your Google Colab!
- Make sure you have uploaded the required CSV files to your google drive
    - Remember the file path!

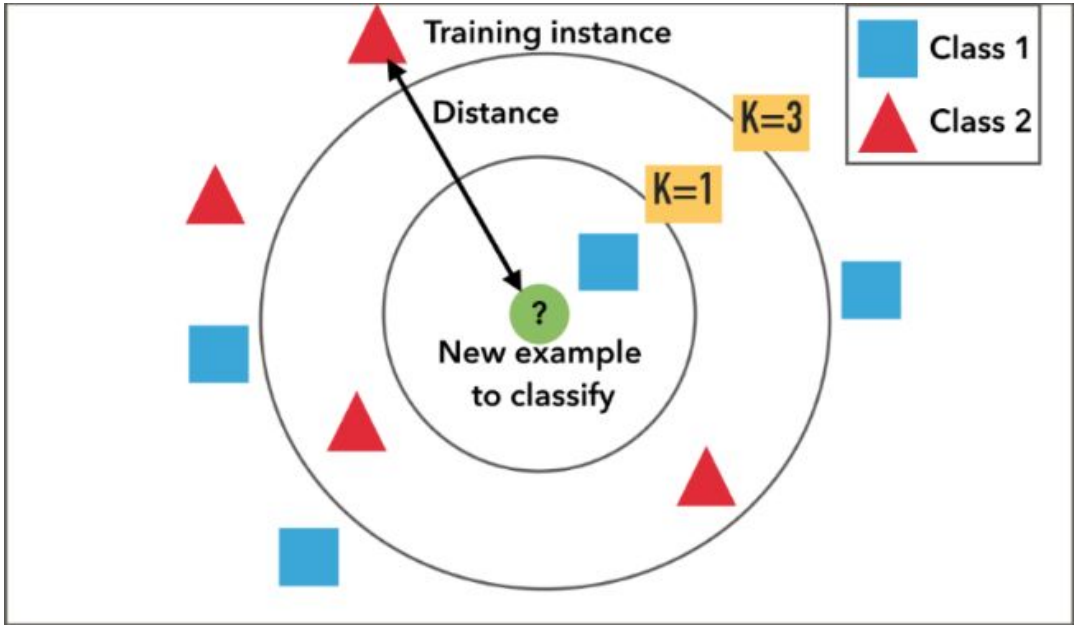


Any
Question

# K-Nearest Neighbors Classifier

- Recall: KNN classifies the label of a data point based on the majority label of its K nearest neighbors

- Naturally, we can tune K
  - i.e. the number of neighbors we want to look at

- Pattern
  - Smaller K: higher variance
    - more likely to overfitting
  - Vice versa



# Tuning KNN

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV

knn_clf = KNeighborsClassifier()
parameters = {
    'n_neighbors': (2,3,4,5,6,7,8)
}
knn_clf_gridcv = GridSearchCV(knn_clf, parameters,
                              cv=5, scoring='accuracy')
knn_clf_gridcv.fit(X_train, y_train)
```

*tested number of considered neighbors*

*Evaluation metric used*

*(Note: can be adjusted based on condition)*

# Tuning KNN

The results

```
# the compact results
cv_result = pd.DataFrame(knn_clf_gridcv.cv_results_)
retain_cols = ['params','mean_test_score','rank_test_score']
cv_result[retain_cols]
```

| | params | mean_test_score | rank_test_score |
|---|---|---|---|
| 0 | {'n_neighbors': 2} | 0.682092 | 6 |
| 1 | {'n_neighbors': 3} | 0.665248 | 7 |
| 2 | {'n_neighbors': 4} | 0.694681 | 4 |
| 3 | {'n_neighbors': 5} | 0.686259 | 5 |
| 4 | {'n_neighbors': 6} | 0.715514 | 1 |
| 5 | {'n_neighbors': 7} | 0.711436 | 2 |
| 6 | {'n_neighbors': 8} | 0.707181 | 3 |

→ *Best model*

# Tuning KNN

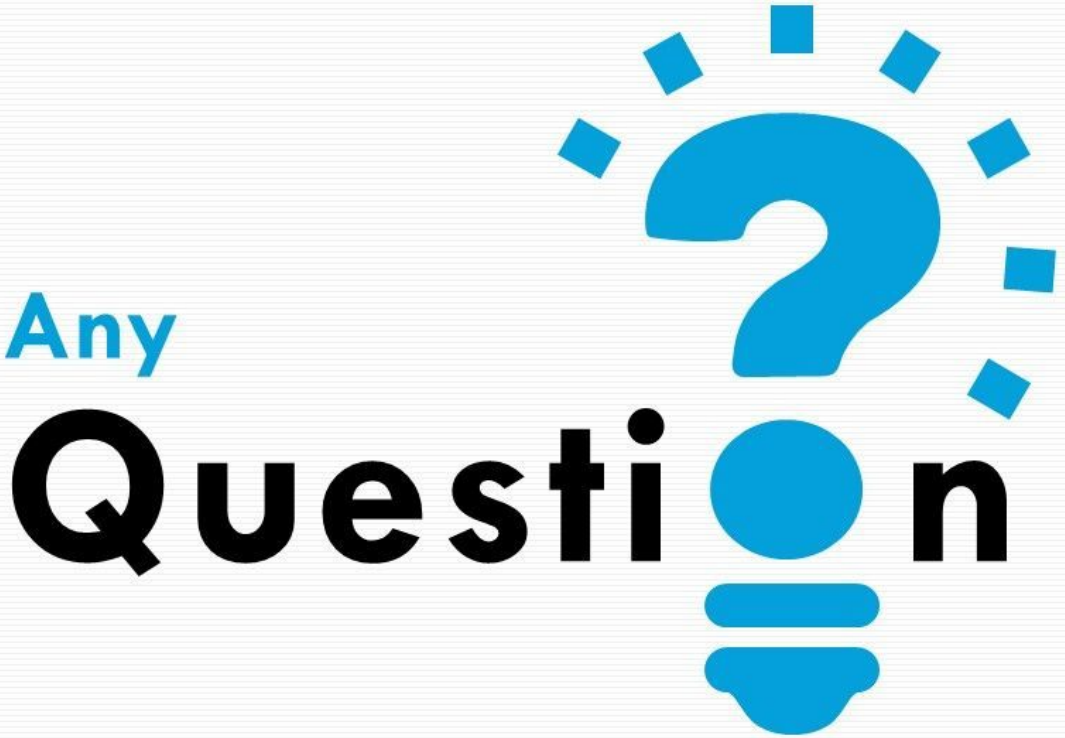
Evaluating the best model obtained

```
# classification report
from sklearn.metrics import classification_report
y_pred = knn_clf_gridcv.best_estimator_.predict(X_test)
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.57      0.80      0.67        35
           1       0.36      0.16      0.22        25

    accuracy                           0.53        60
   macro avg       0.47      0.48      0.44        60
weighted avg       0.48      0.53      0.48        60
```

# Hands-On

- Open today's Jupyter notebook on your Google Colab!
- Make sure you have uploaded the required CSV files to your google drive
    - Remember the file path!

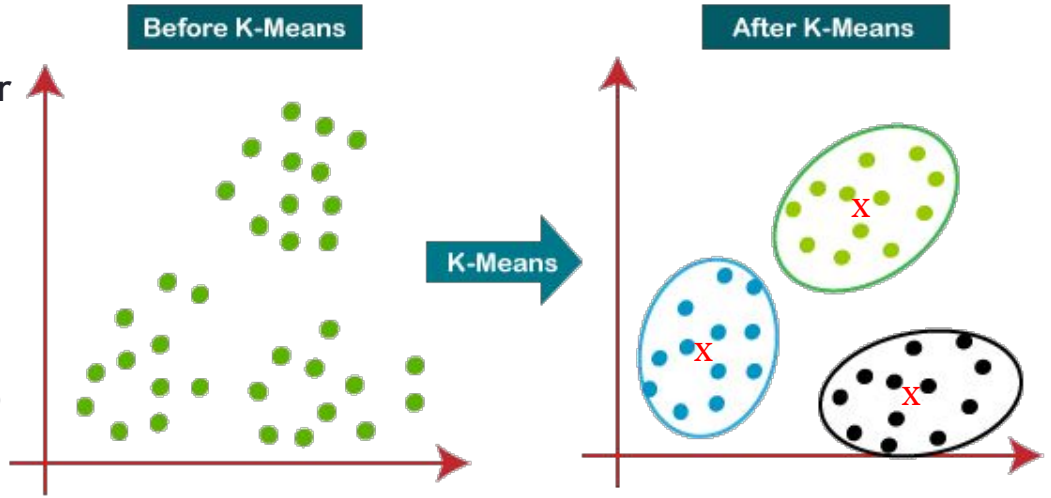


Any
Question

# K-Means Clustering

- Recall: K Means algorithm is a unsupervised model to find K similar groups in data

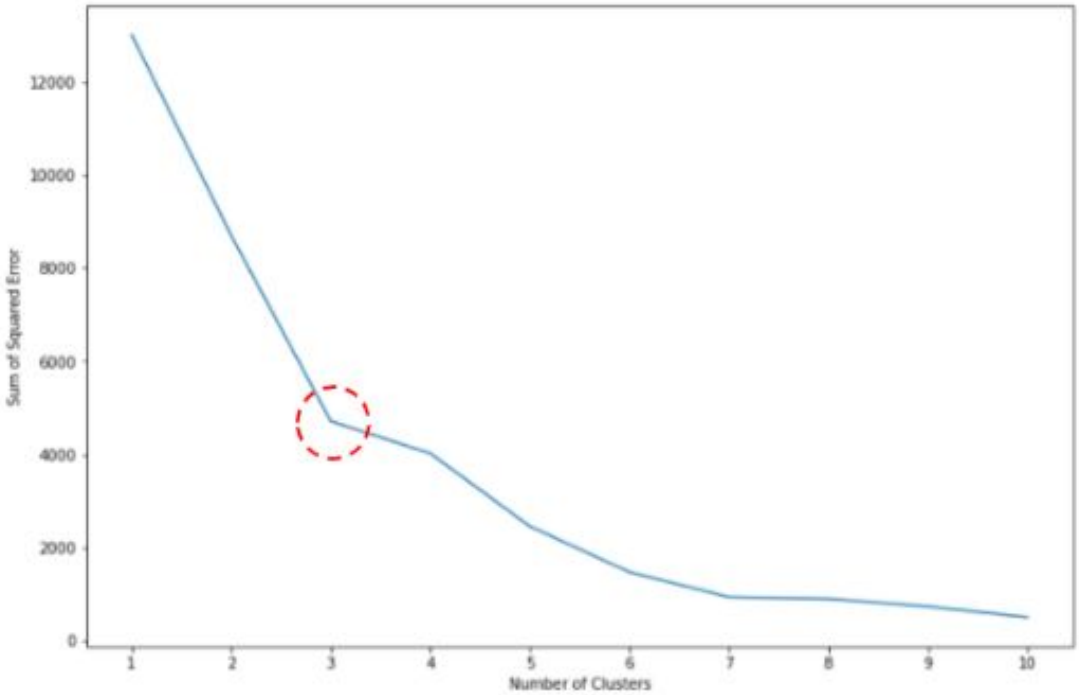
- The relevant metric is **Sum of Squared Error**
  - The “error” refers to the deviation/distance between a data point and its centroid.

- Naturally, we can tune K
  - i.e. the number of clusters we want to have



# Tuning K-Means Algorithm

- NO, we won't use GridSearchCV
- Instead, we will use the **Elbow Method**
    - We plot the sum of squared errors (SSE) for various K values
    - Choose the K value at which the SSE decline slopes change significantly (forming an "elbow" shape)



# Tuning K-Means Algorithm

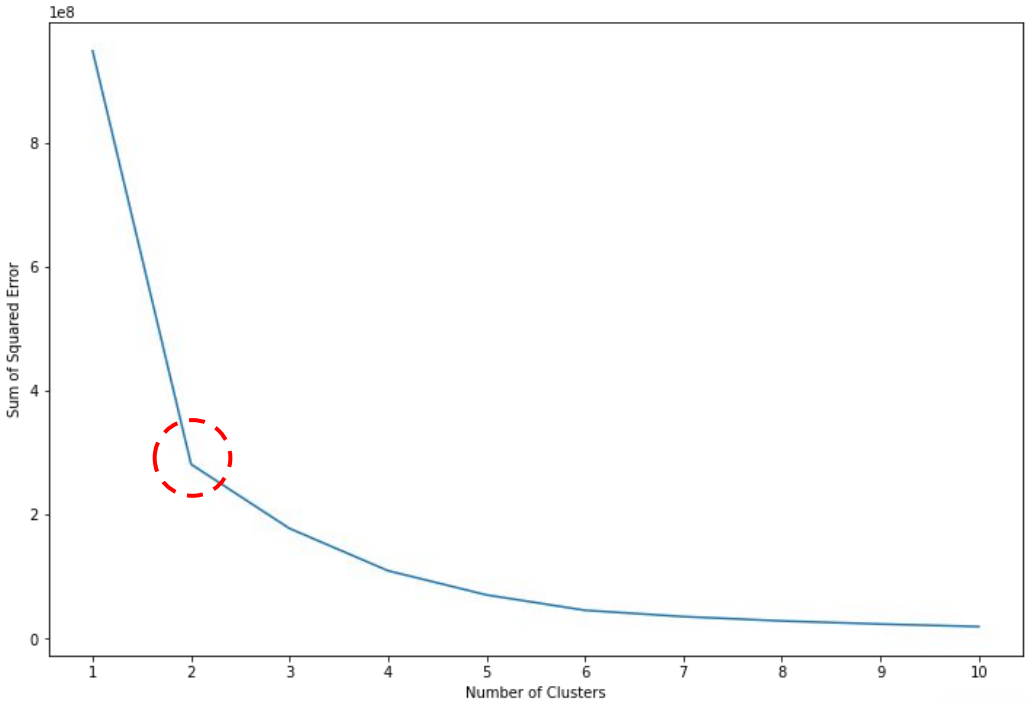
```
from sklearn.cluster import KMeans

# track sum of squared error
sse = []
for k in range(1,11):
    kmeans = KMeans(n_clusters=k)
    kmeans.fit(X)
    sse.append(kmeans.inertia_)
```

*tested k = [1,2,3,..., 10]*

*Sum of Squared Error*

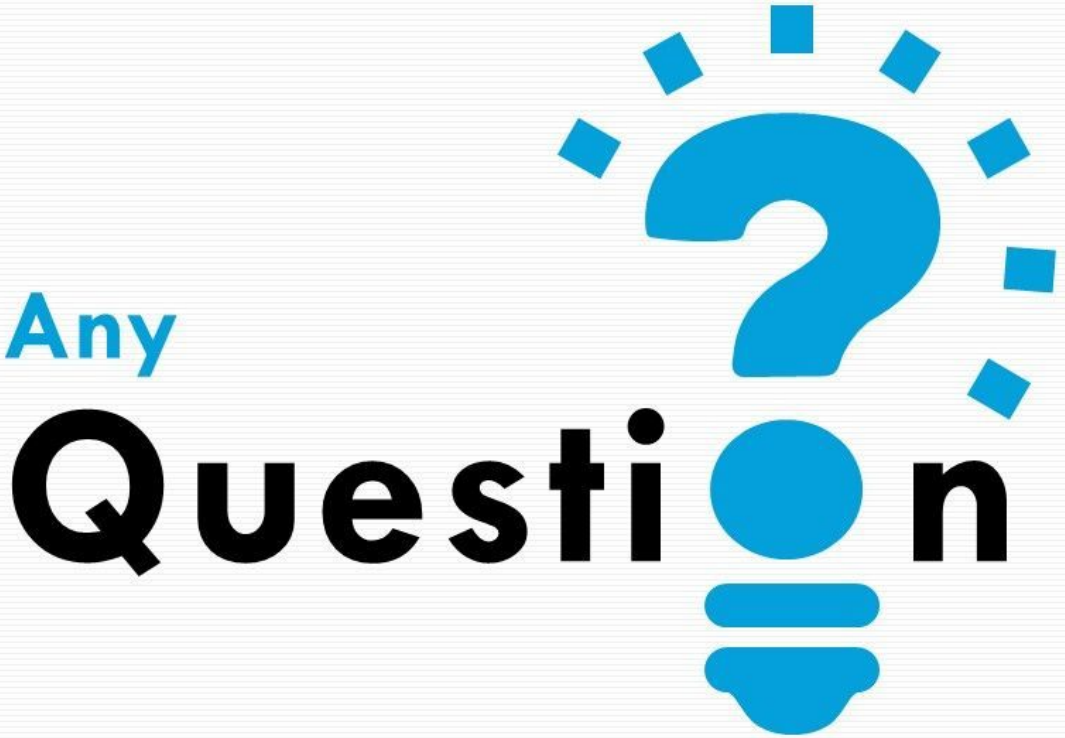
# Tuning K-Means Algorithm

```
# draw the SSE decline progression
import matplotlib.pyplot as plt
plt.figure(figsize = (12,8))
plt.plot(range(1,11), sse)
plt.xticks(range(1,11))
plt.xlabel("Number of Clusters")
plt.ylabel("Sum of Squared Error")
plt.show()
```



# Hands-On

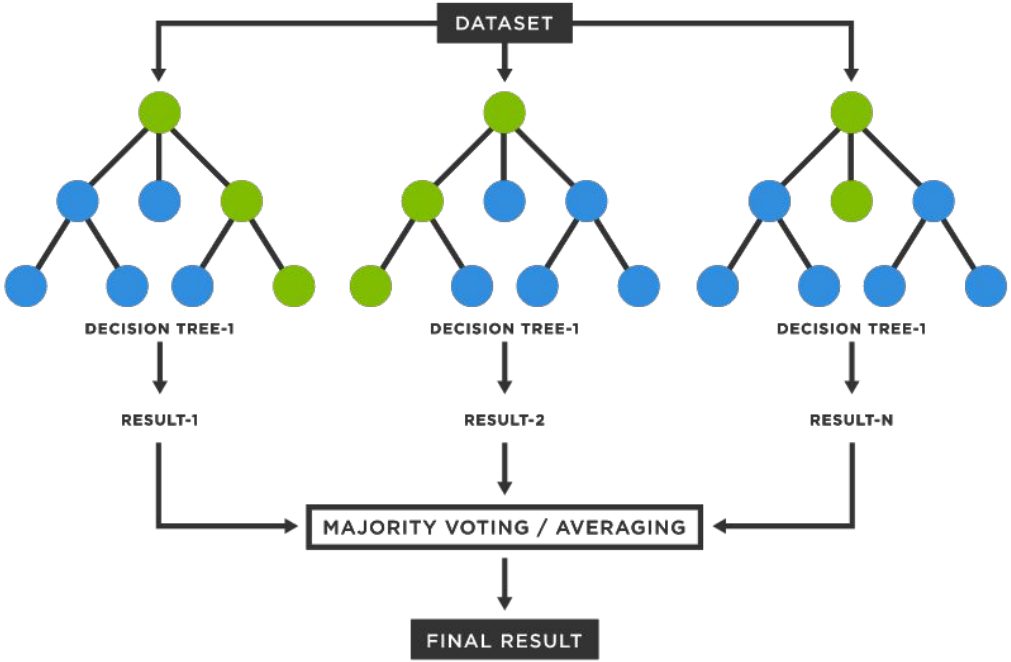
- Open today's Jupyter notebook on your Google Colab!
- Make sure you have uploaded the required CSV files to your google drive
    - Remember the file path!



Any
Question

# Random Forest

- Recall: Random Forest is a group of decision trees that predict target variable **by taking majority votes** among them

- There are several things that can be optimized:
  - number of trees in the “forest”
  - max tree depth
  - max number of leafs for each tree
  - etc



# Tuning Random Forest

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV

rf_clf = RandomForestClassifier(random_state=42)

parameters = {
    'n_estimators': (10,20,30,40,50),
    'max_depth':(1,2,3,4,5)
}

# Cross Validation
rf_clf_gridcv = GridSearchCV(rf_clf, parameters, cv=5,
                             scoring='recall')
rf_clf_gridcv.fit(X_train, y_train)
```

→ *Hyperparam 1: number of trees*

→ *Hyperparam 2: max tree depth*

→ *Evaluation metric used*

# Tuning Random Forest

The results

```
# the results
cv_result = pd.DataFrame(rf_clf_gridcv.cv_results_)
retain_cols = ['params','mean_test_score','rank_test_score']
cv_result[retain_cols].sort_values('rank_test_score')
```

→ *Sorted from the best model*

| | params | mean_test_score | rank_test_score |
|---|---|---|---|
| **12** | {'max_depth': 3, 'n_estimators': 30} | 0.761905 | 1 |
| **24** | {'max_depth': 5, 'n_estimators': 50} | 0.746667 | 2 |
| **11** | {'max_depth': 3, 'n_estimators': 20} | 0.745714 | 3 |
| **22** | {'max_depth': 5, 'n_estimators': 30} | 0.732381 | 4 |
| **19** | {'max_depth': 4, 'n_estimators': 50} | 0.719048 | 5 |
| **17** | {'max_depth': 4, 'n_estimators': 30} | 0.719048 | 5 |
| **23** | {'max_depth': 5, 'n_estimators': 40} | 0.718095 | 7 |

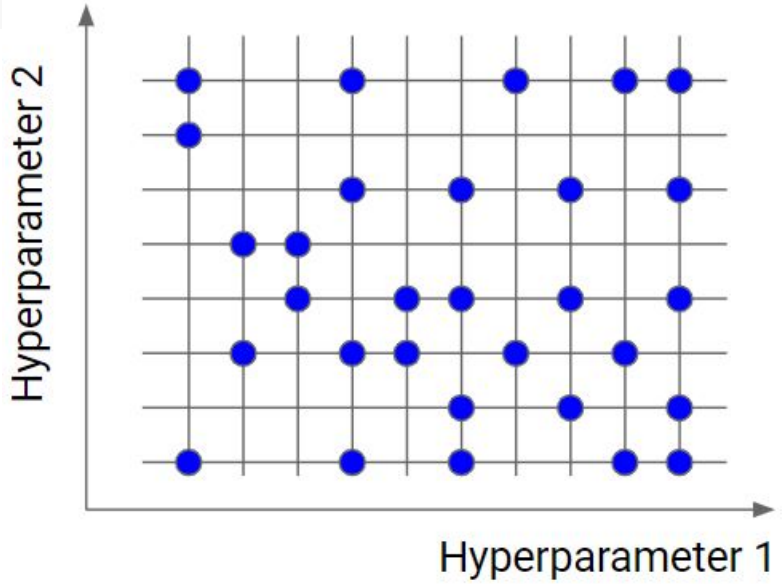
# RandomizedSearchCV

```
parameters = {
    'n_estimators': (10,20,30,40,50),
    'max_depth':(1,2,3,4,5)
}
```

There are 5 x 5 = 25 combinations!
Can be long to run (compute)!

In such cases, we may use **RandomizedSearchCV**

- It only considers a **subset of random combinations** of hyperparameter values



# Tuning Random Forest

```
# using random search CV
from sklearn.model_selection import RandomizedSearchCV

parameters = {
    'n_estimators': (10,20,30,40,50),
    'max_depth':(1,2,3,4,5)
}

rf_clf_randomcv = RandomizedSearchCV(rf_clf, parameters, cv=5,
                                     scoring='recall', n_iter=10)
rf_clf_randomcv.fit(X_train, y_train)
```

→ *Number of pairs tested*

# Tuning Random Forest

The results using RandomizedSearchCV

```
cv_result = pd.DataFrame(rf_clf_randomcv.cv_results_)
retain_cols = ['params','mean_test_score','rank_test_score']
cv_result[retain_cols].sort_values('rank_test_score')
```

| | params | mean_test_score | rank_test_score |
|---|---|---|---|
| **0** | {'n_estimators': 30, 'max_depth': 3} | 0.761905 | 1 |
| **3** | {'n_estimators': 50, 'max_depth': 4} | 0.719048 | 2 |
| **5** | {'n_estimators': 30, 'max_depth': 4} | 0.719048 | 2 |
| **9** | {'n_estimators': 10, 'max_depth': 5} | 0.689524 | 4 |
| **6** | {'n_estimators': 50, 'max_depth': 3} | 0.648571 | 5 |
| **2** | {'n_estimators': 30, 'max_depth': 2} | 0.620952 | 6 |
| **7** | {'n_estimators': 50, 'max_depth': 2} | 0.591429 | 7 |

→ *Same as GridSearchCV*

*Note that this is NOT always the case*

# Tuning Random Forest

Evaluating the best model obtained

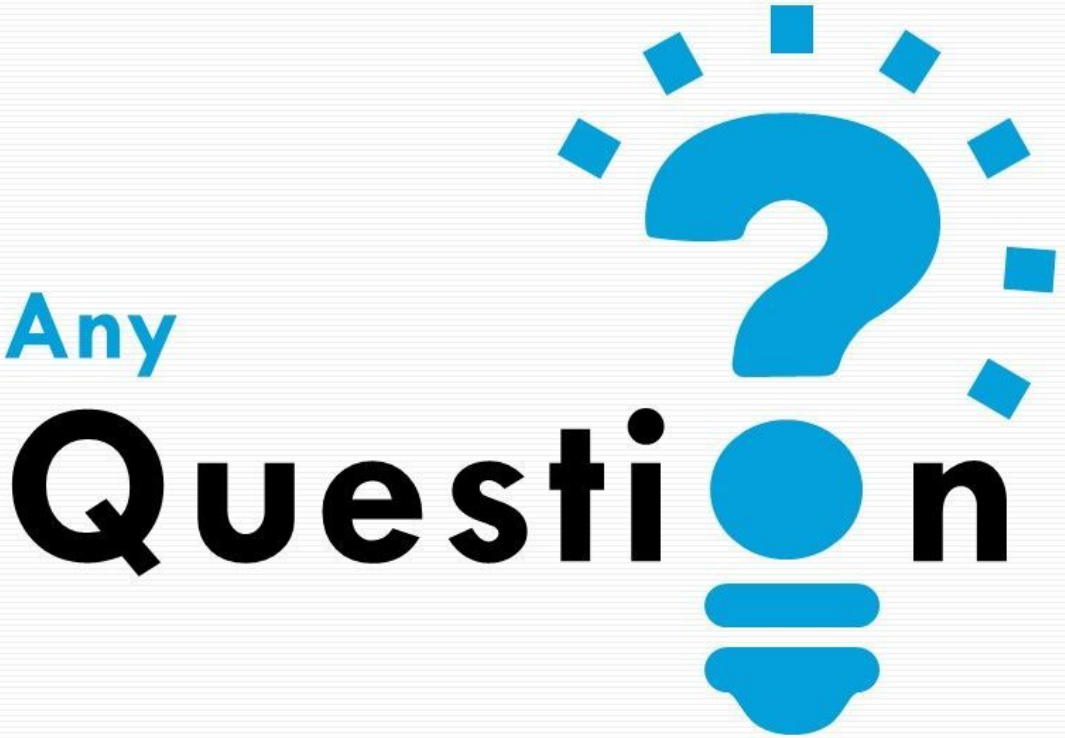
```
# classification report
from sklearn.metrics import classification_report
y_pred = rf_clf_randomcv.best_estimator_.predict(X_test)
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.73      0.94      0.83        35
           1       0.87      0.52      0.65        25

    accuracy                           0.77        60
   macro avg       0.80      0.73      0.74        60
weighted avg       0.79      0.77      0.75        60
```

→ *Better than KNN*

# Hands-On

- Open today's Jupyter notebook on your Google Colab!
- Make sure you have uploaded the required CSV files to your google drive
    - Remember the file path!



Any
Question

# Assignment

- What to submit? Google colab link (don’t forget to share access to me:  pararawendy19@gmail.com)
    - Format notebook name: HW_HPTUNING_<YOUR COMPLETE NAME>

# Problem

- Load churn.csv as DataFrame
- Basic data cleaning (missing values, duplicates) (10 points)
- Split the data: training & testing *(10 points)*
- Multicollinearity study *(10 points)*
    - And feature selection (if any)
    - *Recall: the threshold is at least 0.8 (absolute value)*
- Handle categorical data *(20 points)*
    - Column with 2 distinct values → convert to binary numeric {0,1}
    - Else → One Hot Encode
- Choose the appropriate metric for fitting the model *(10 points)*
- Train any classification model you'd prefer *(30 points)*
    - YET please ensure you do hyperparameter tuning
- Evaluate the model on test data *(10 points)*

# Thank you