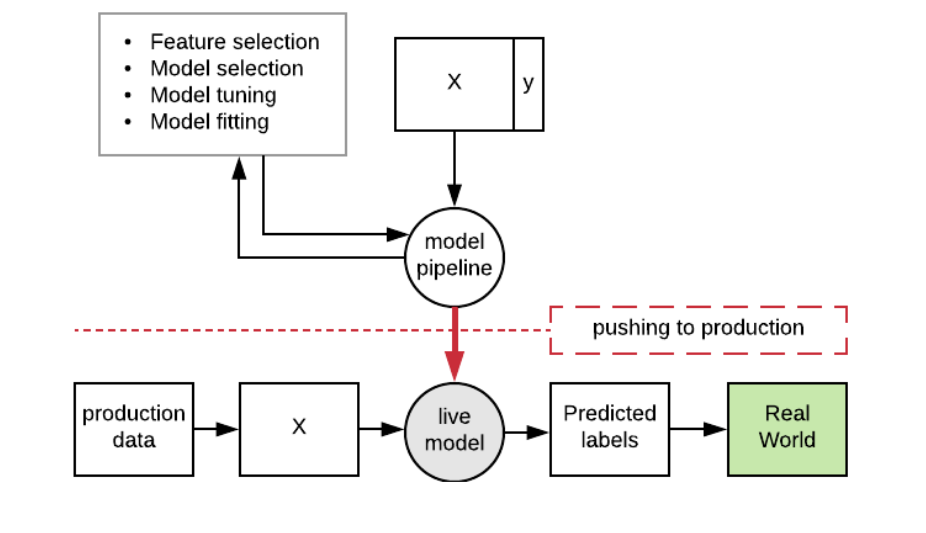
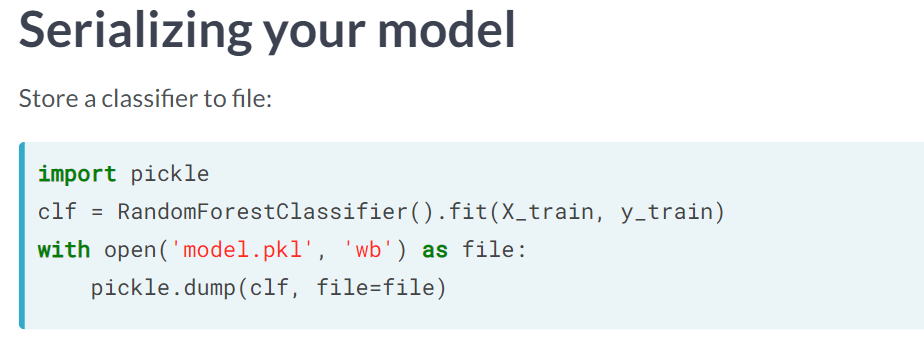
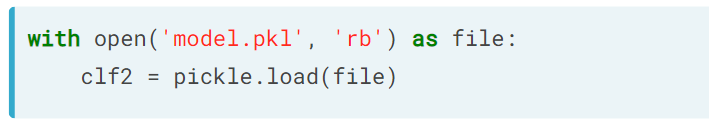
***Designing Machine Learning Workflows***

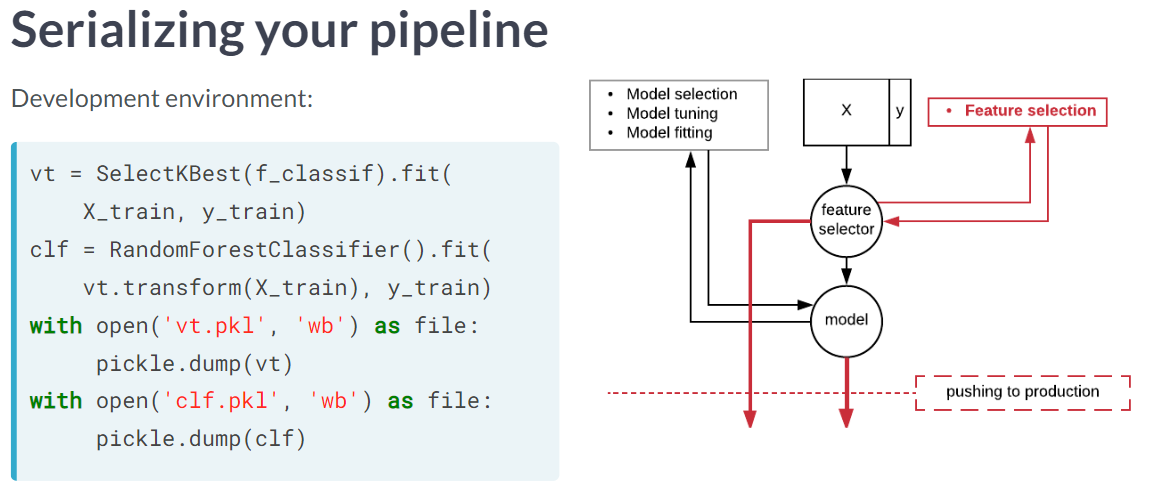




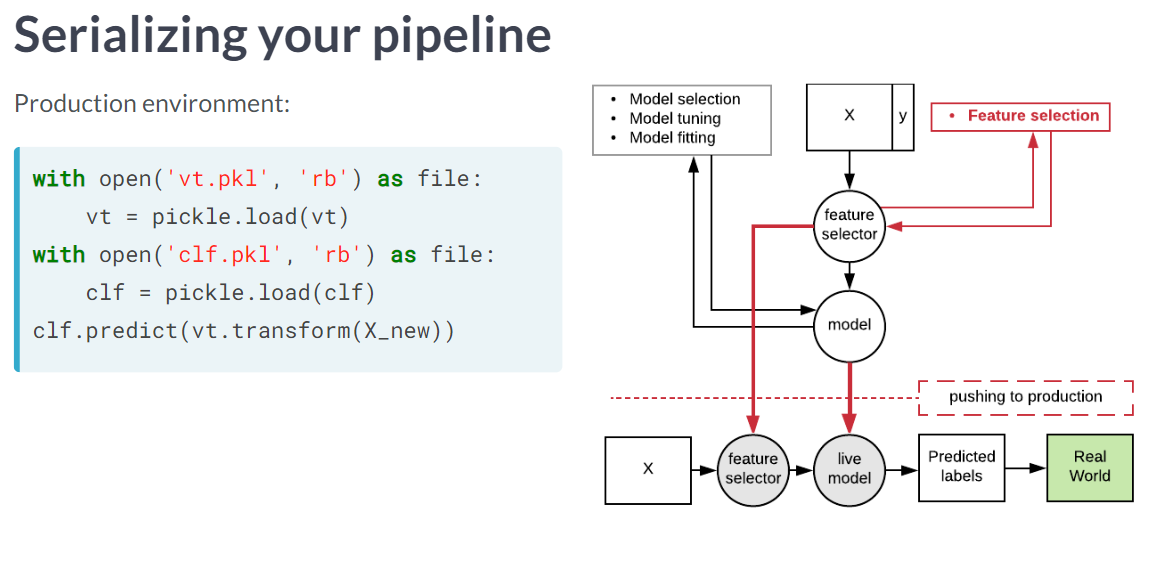
Wb -> write Binary

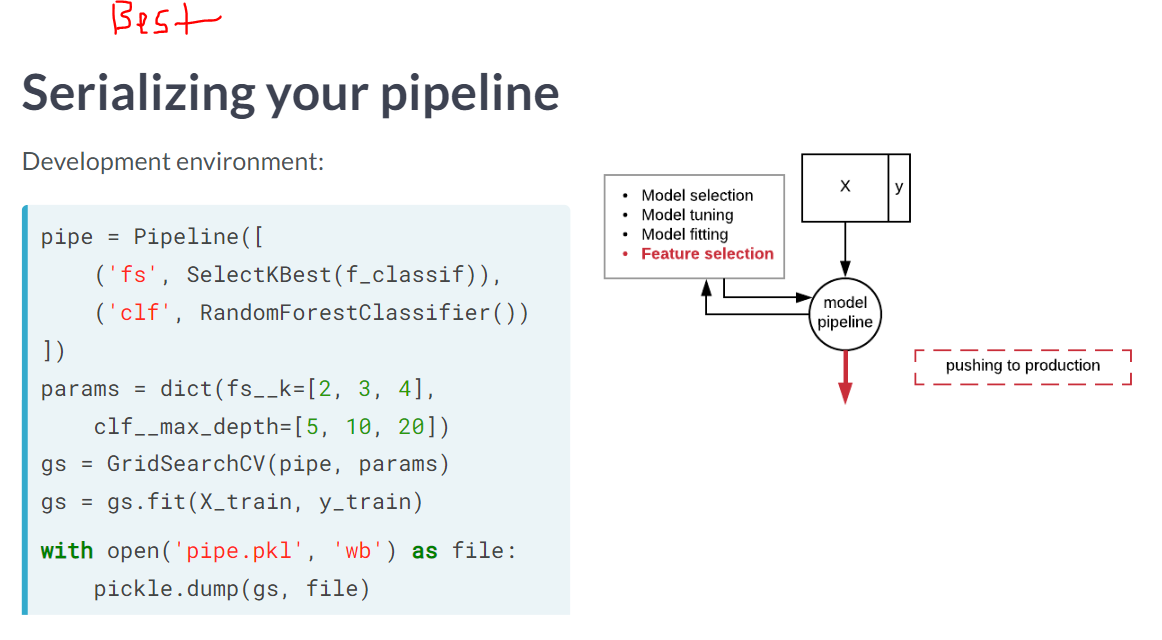


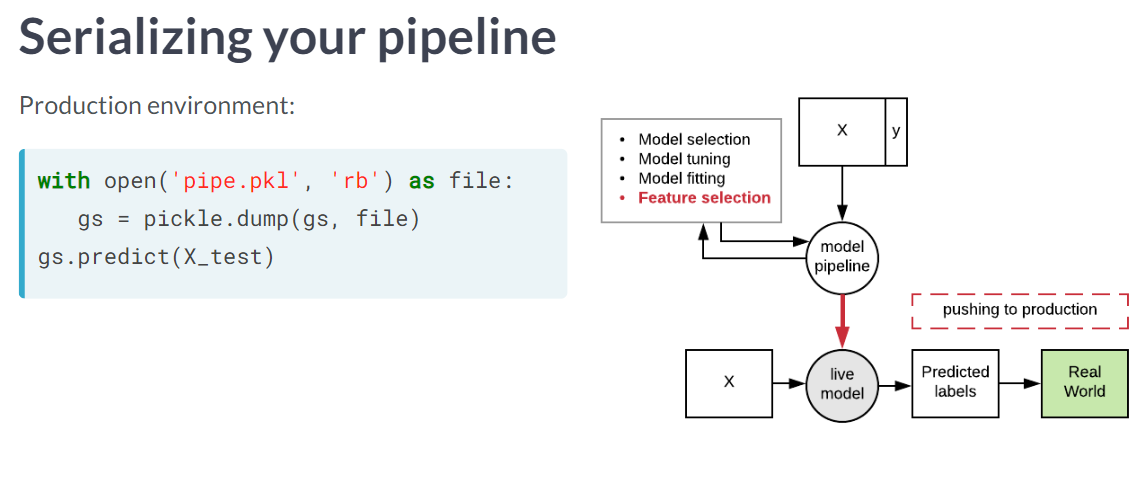
**Extract Feature selector / feature transformation code and feed into classifier**



To Transform the new data using feature selector:







# Pickles

# Fit a random forest to the training set

clf = RandomForestClassifier(random\_state=42)

clf.fit(X\_train, y\_train)

# Save it to a file, to be pushed to production

with open('model.pkl', 'wb') as file:

pickle.dump(clf, file=file)

# Now load the model from file in the production environment

with open('model.pkl','rb') as file:

clf\_from\_file = pickle.load(file)

# Predict the labels of the test dataset

preds = clf\_from\_file.predict(X\_test)

**Custom function transformers in pipelines**

# Define a feature extractor to flag very large values

def more\_than\_average(X, multiplier=1.0):

Z = X.copy()

Z[:,1] = Z[:,1] > multiplier\*np.mean(Z[:,1])

return Z

# Convert your function so that it can be used in a pipeline

pipe = Pipeline([

('ft', FunctionTransformer(more\_than\_average)),

('clf', RandomForestClassifier(random\_state=2))])

# Optimize the parameter multiplier using GridSearchCV

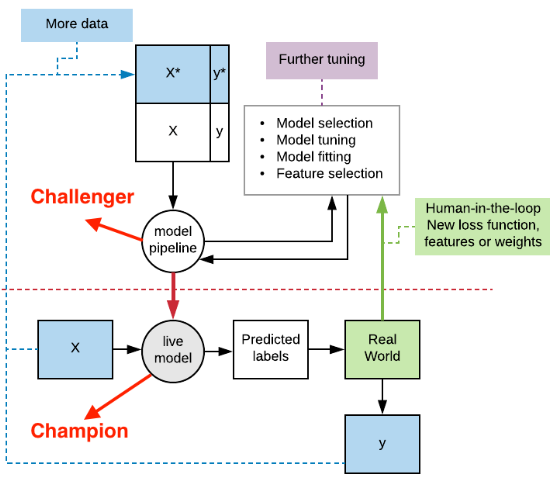
params = {'ft\_\_multiplier':[1, 2, 3]}

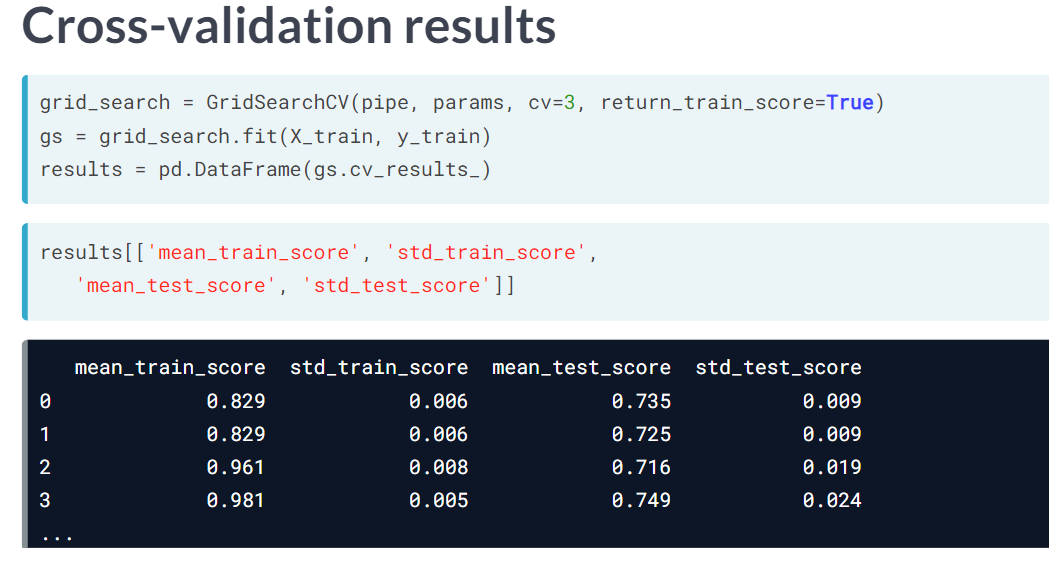
grid\_search = GridSearchCV(pipe, param\_grid=params)

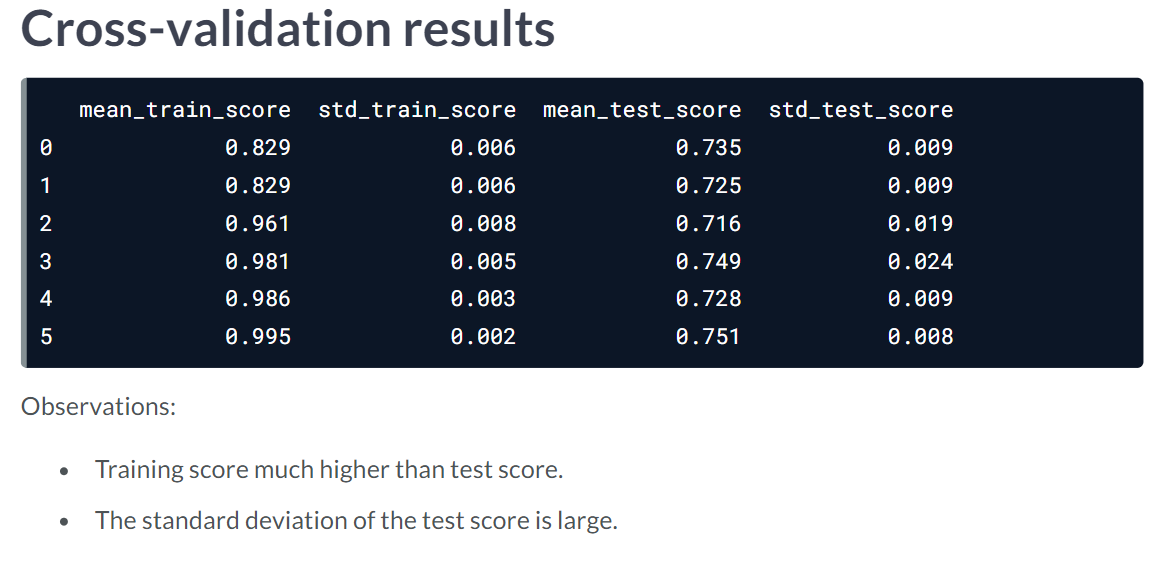
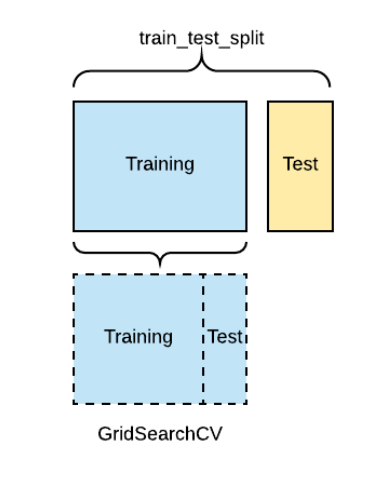
# Agile Software Development practice for Iterating without overfitting

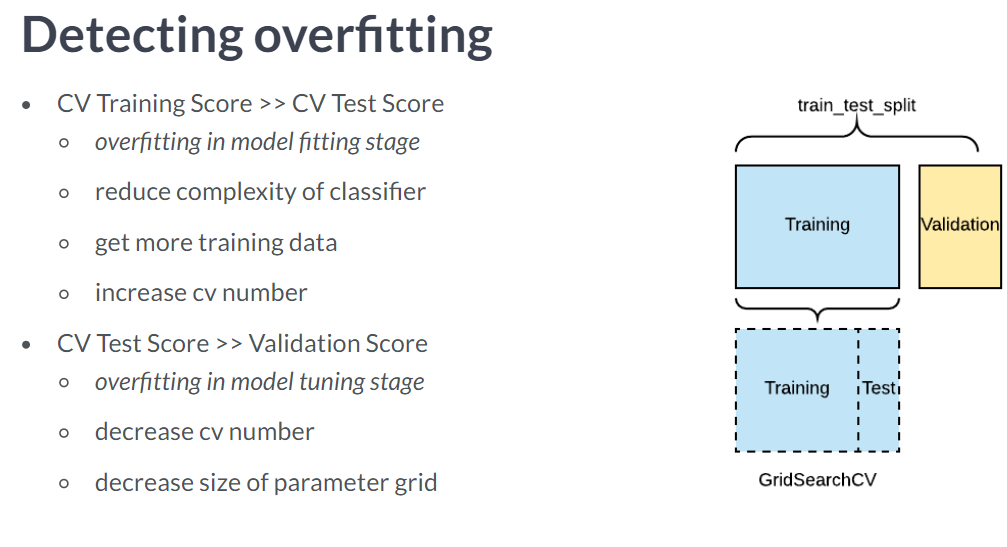
Here we need not build the Model again and deploy it to the production, So we can have the pipeline as challenger and the model deployed as champion, when we change the pipeline in the development environment, automatically it affects the Production Model.

For example when we get more data, or we decide to remove or include columns etc…

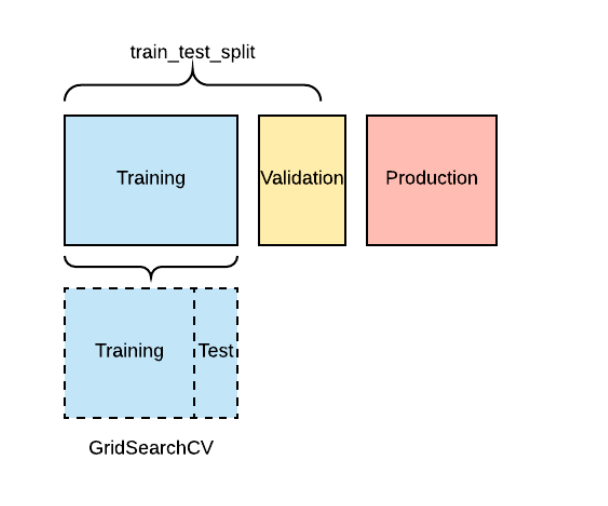








**When we get new data in Production (Dataset Shift)**



# Challenge the champion

Having pushed your random forest to production, you suddenly worry that a naive Bayes classifier might be better. You want to run a champion-challenger test, by comparing a naive Bayes, acting as the challenger, to exactly the model which is currently in production, which you will load from file to make sure there is no confusion. You will use the F1 score for assessment. You have the data X\_train, X\_test, y\_train and y\_test available as before and GaussianNB(), f1\_score() and pickle().  
  
**# Load the current model from disk**

champion = pickle.load(open('model.pkl', 'rb'))

**# Fit a Gaussian Naive Bayes to the training data**

challenger = GaussianNB().fit(X\_train, y\_train)

**# Print the F1 test scores of both champion and challenger**

print(f1\_score(y\_test, champion.predict(X\_test)))

print(f1\_score(y\_test, challenger.predict(X\_test)))

**# Write back to disk the best-performing model**

with open('model.pkl', 'wb') as file:

pickle.dump(champion, file=file)

# Cross-validation statistics

You used grid search CV to tune your random forest classifier, and now want to inspect the cross-validation results to ensure you did not overfit. In particular you would like to take the difference of the mean test score for each fold from the mean training score. The dataset is available as X\_train and y\_train, the pipeline as pipe, and a number of modules are pre-loaded including pandas as pd and GridSearchCV().

# Fit your pipeline using GridSearchCV with three folds

grid\_search = GridSearchCV(

pipe, params, cv=3, return\_train\_score=True)

# Fit the grid search

gs = grid\_search.fit(X\_train, y\_train)

# Store the results of CV into a pandas dataframe

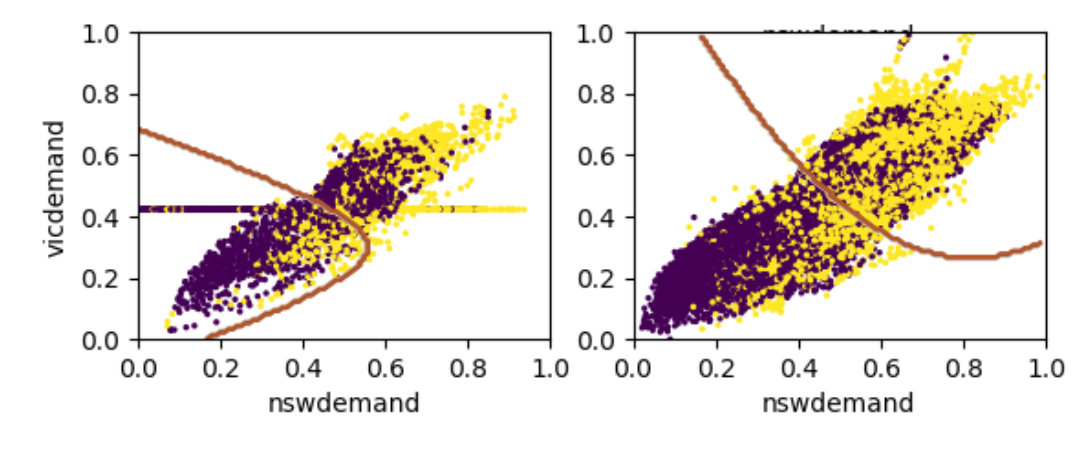
results = pd.DataFrame(gs.cv\_results\_)

# Print the difference between mean test and training scores

print(

results['mean\_test\_score']-results['mean\_train\_score'])

# Dataset shift / Concept Drift



**How to protect the model against dataset shit is to Regularly retrain it using only a batch of recent data, a technique known as windowing.**



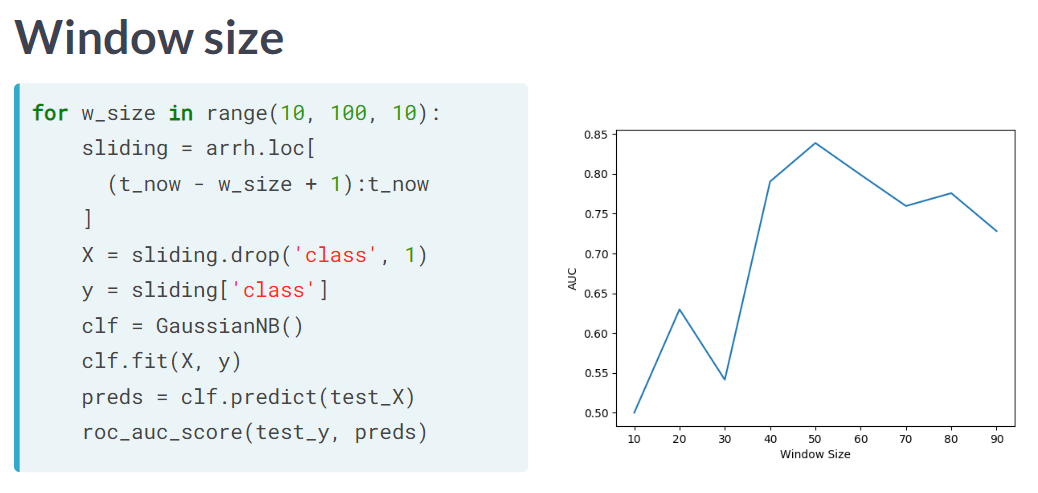
**We can detect dataset shit by comparing champion and challenger setup models, expanding window contain more data than sliding windows, so the only reason for the later to win is the dataset shift.**

Train a classifier on time after 4000 and last 20000 data points

We will use all data after 4000 as test data



Comparing AUC suggests that sliding window outperforms the expanding one, so there is a dataset shift



Good practice to fit the data on different window size and pick which size it performs the best.

Here 50 seems best

# Tuning the window size

You want to check for yourself that the optimal window size for the arrhythmia dataset is 50. You have been given the dataset as a pandas data frame called arrh, and want to use a subset of the data up to time t\_now. Your test data is available as X\_test, y\_test. You will try out a number of window sizes, ranging from 10 to 100, fit a naive Bayes classifier to each window, assess its F1 score on the test data, and then pick the best performing window size. You also have numpy available as np, and the function f1\_score() has been imported already. Finally, an empty list called accuracies has been initialized for you to store the accuracies of the windows.

* Define the index of a sliding window of size w\_size stopping at t\_now using the .loc() method.
* Construct X from the sliding window by removing the class column. Store that latter column as y.
* Fit a naive Bayes classifier to X and y, and use it to predict the labels of the test data X\_test.
* Compute the F1 score of these predictions for each window size, and find the best-performing window size.

# Loop over window sizes

for w\_size in wrange:

# Define sliding window

sliding = arrh.loc[(t\_now - w\_size + 1):t\_now]

# Extract X and y from the sliding window

X, y = sliding.drop('class', 1), sliding['class']

# Fit the classifier and store the F1 score

preds = GaussianNB().fit(X, y).predict(X\_test)

accuracies.append(f1\_score(y\_test, preds))

# Estimate the best performing window size

optimal\_window = wrange[np.argmax(accuracies)]

**Bringing it all together**

You have two concerns about your pipeline at the arrhythmia detection startup:

* The app was trained on patients of all ages, but is primarily being used by fitness users who tend to be young. You suspect this might be a case of domain shift, and hence want to disregard all examples above 50 years old.
* You are still concerned about overfitting, so you want to see if making the random forest classifier less complex and selecting some features might help with that.

You will create a pipeline with a feature selection SelectKBest() step and a RandomForestClassifier, both of which have been imported. You also have access to GridSearchCV(), Pipeline, numpy as np and pickle. The data is available as arrh.

* Create a pipeline with SelectKBest() as step ft and RandomForestClassifier() as step clf.
* Create a parameter grid to tune k in SelectKBest() and max\_depth in RandomForestClassifier().
* Use GridSearchCV() to optimize your pipeline against that grid and data containing only those aged under 50.
* Save the optimized pipeline to a pickle for production.

# Create a pipeline

pipe = Pipeline([

('ft', SelectKBest()), ('clf', RandomForestClassifier(random\_state=2))])

# Create a parameter grid

grid = {'ft\_\_k':[5, 10], 'clf\_\_max\_depth':[10, 20]}

# Execute grid search CV on a dataset containing under 50s

grid\_search = GridSearchCV(pipe, param\_grid=grid)

arrh = arrh.iloc[np.where(arrh['age'] < 50)]

grid\_search.fit(arrh.drop('class', 1), arrh['class'])

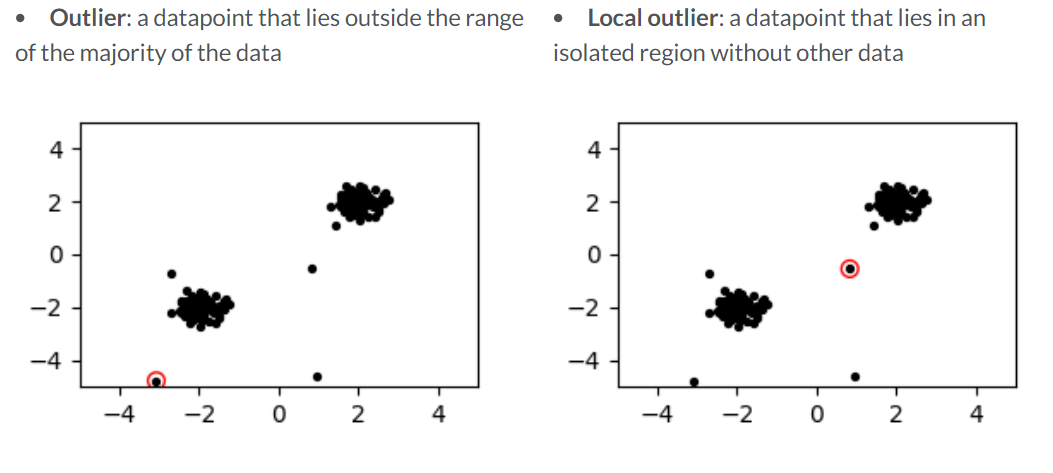
# Push the fitted pipeline to production

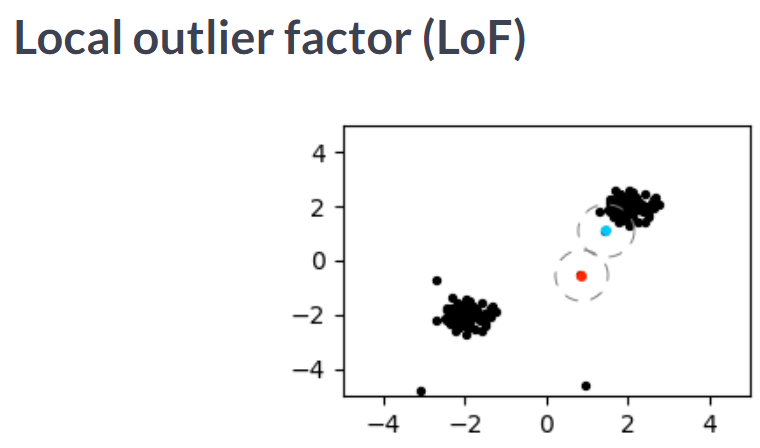
with open('pipe.pkl', 'wb') as file:

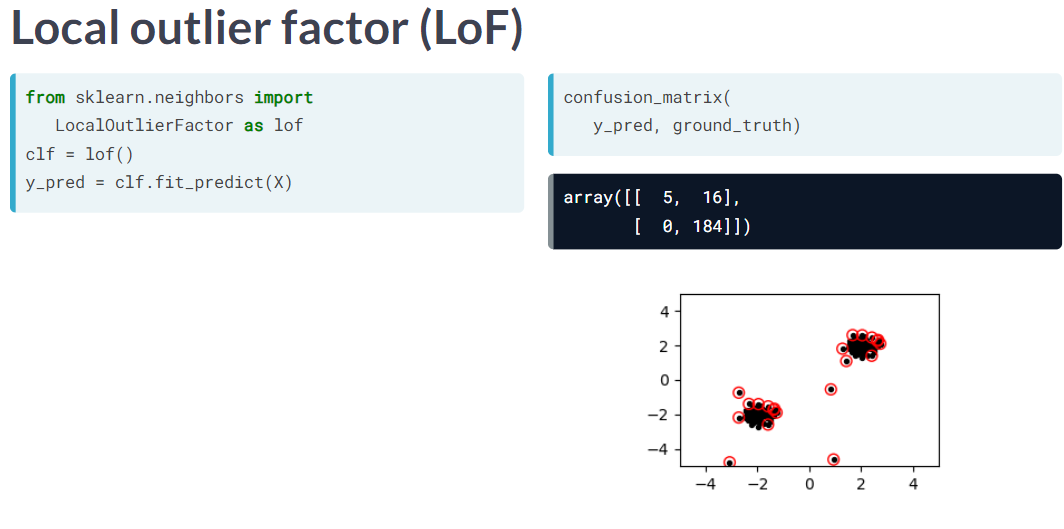
pickle.dump(grid\_search, file)

**Anomaly Detection:**

When there is a challenge of modeling data without any, or with very few, labels. This takes you into a journey into anomaly detection, a kind of unsupervised modeling, as well as distance-based learning, where beliefs about what constitutes similarity between two examples can be used in place of labels to help you achieve levels of accuracy comparable to a supervised workflow.

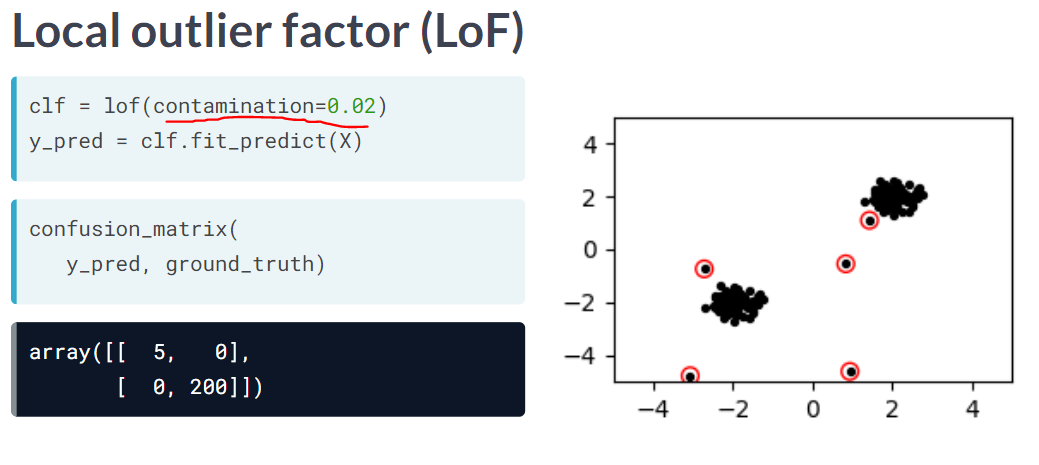






We can tune some threshold for false positive rates by taking input from the domain expert

What percent of data is anomalous ? ask user and include it in PARAM as contamination



# Novelty detection

Detect Anomalies in future data?

