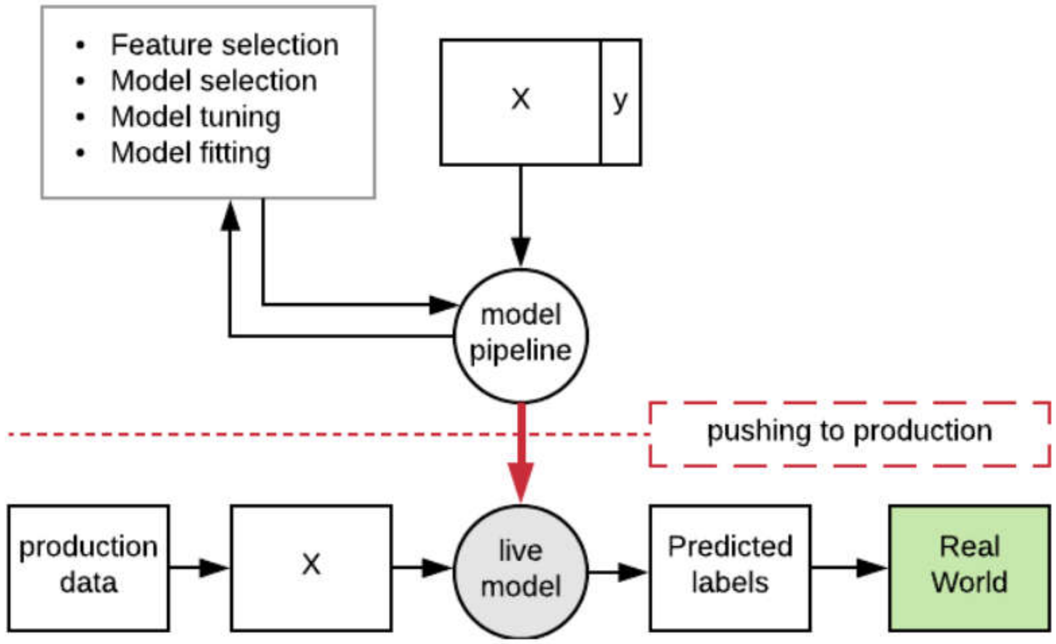
Designing Machine Learning Workflows



Serializing your model

Store a classifier to file:

```python
import pickle
clf = RandomForestClassifier().fit(X_train, y_train)
with open('model.pkl', 'wb') as file:
    pickle.dump(clf, file=file)
```

Wb -> write Binary

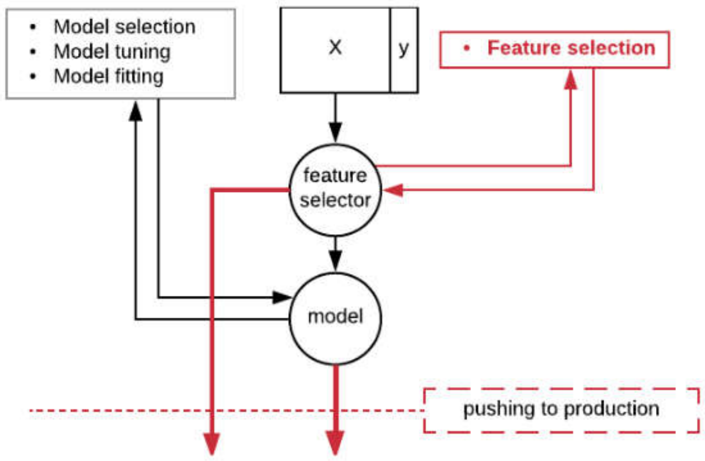
```python
with open('model.pkl', 'rb') as file:
    clf2 = pickle.load(file)
```

Extract Feature selector / feature transformation code and feed into classifier

Serializing your pipeline

Development environment:

```python
vt = SelectKBest(f_classif).fit(
    X_train, y_train)
clf = RandomForestClassifier().fit(
    vt.transform(X_train), y_train)
with open('vt.pkl', 'wb') as file:
     pickle.dump(vt)
with open('clf.pkl', 'wb') as file:
     pickle.dump(clf)
```

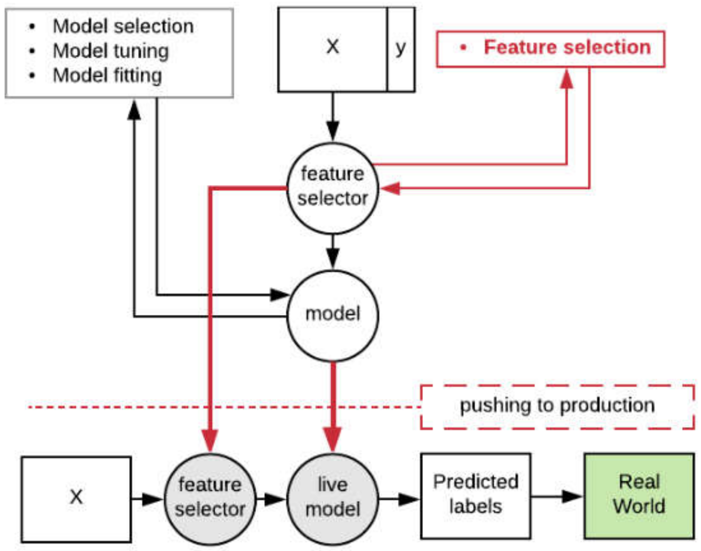


To Transform the new data using feature selector:

Serializing your pipeline

Production environment:

```python
with open('vt.pkl', 'rb') as file:
    vt = pickle.load(vt)
with open('clf.pkl', 'rb') as file:
    clf = pickle.load(clf)
clf.predict(vt.transform(X_new))
```

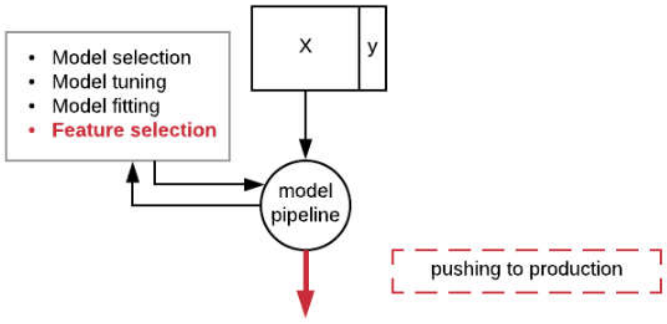


Best

Serializing your pipeline

Development environment:

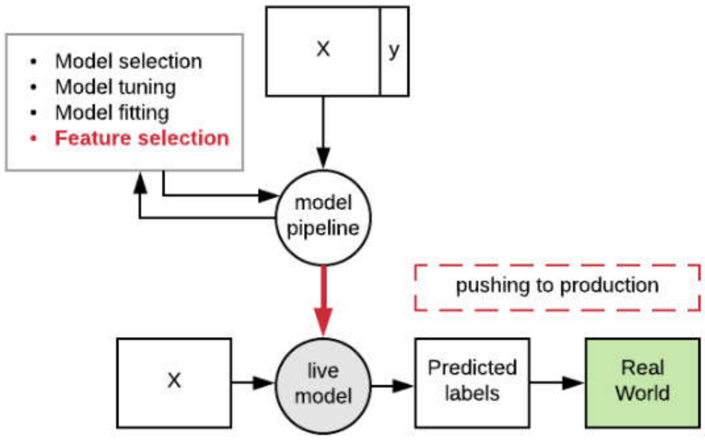
```
pipe = Pipeline([
    ('fs', SelectKBest(f_classif)),
    ('clf', RandomForestClassifier())
])
params = dict(fs__k=[2, 3, 4],
    clf__max_depth=[5, 10, 20])
gs = GridSearchCV(pipe, params)
gs = gs.fit(X_train, y_train)
with open('pipe.pkl', 'wb') as file:
    pickle.dump(gs, file)
```



Serializing your pipeline

Production environment:

```
with open('pipe.pkl', 'rb') as file:
   gs = pickle.dump(gs, file)
gs.predict(X_test)
```



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

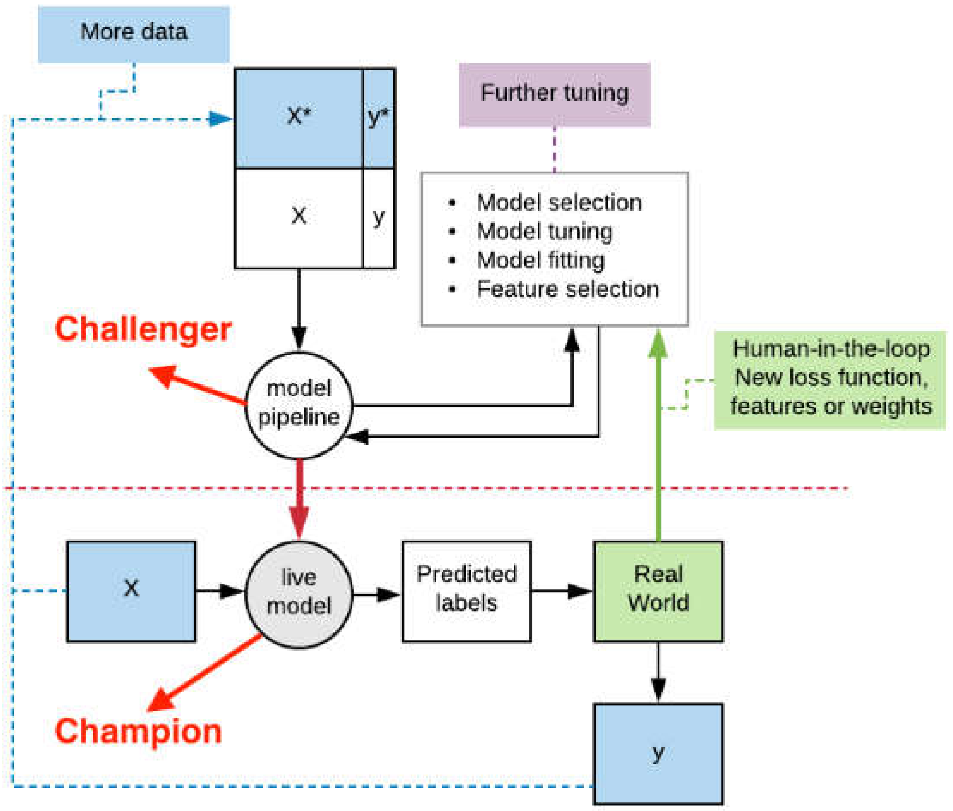
```
# Define a feature extractor to flag very large values
def more_than_average(X, multiplier=1.0):
  Z = X.copy()
  Z[:,1] = Z[:,1] > multiplier*np.mean(Z[:,1])
  return Z

# 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…

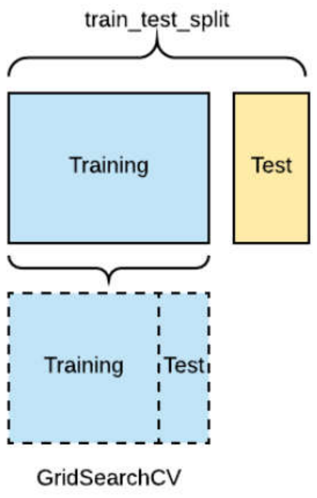


Cross-validation results

```
grid_search = GridSearchCV(pipe, params, cv=3, return_train_score=True)
gs = grid_search.fit(X_train, y_train)
results = pd.DataFrame(gs.cv_results_)
```

```
results[['mean_train_score', 'std_train_score',
   'mean_test_score', 'std_test_score']]
```

```
   mean_train_score  std_train_score  mean_test_score  std_test_score
0             0.829            0.006            0.735           0.009
1             0.829            0.006            0.725           0.009
2             0.961            0.008            0.716           0.019
3             0.981            0.005            0.749           0.024
...
```



Cross-validation results

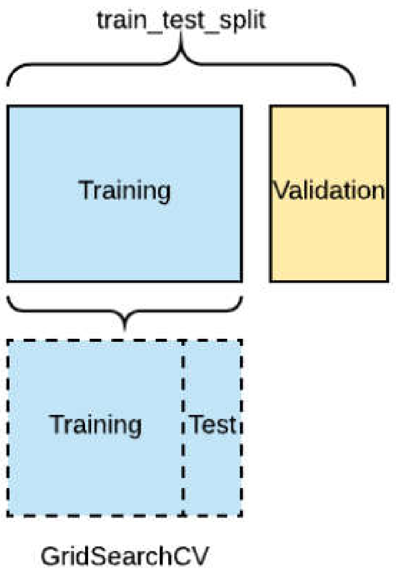
```
   mean_train_score  std_train_score  mean_test_score  std_test_score
0             0.829            0.006            0.735           0.009
1             0.829            0.006            0.725           0.009
2             0.961            0.008            0.716           0.019
3             0.981            0.005            0.749           0.024
4             0.986            0.003            0.728           0.009
5             0.995            0.002            0.751           0.008
```

Observations:

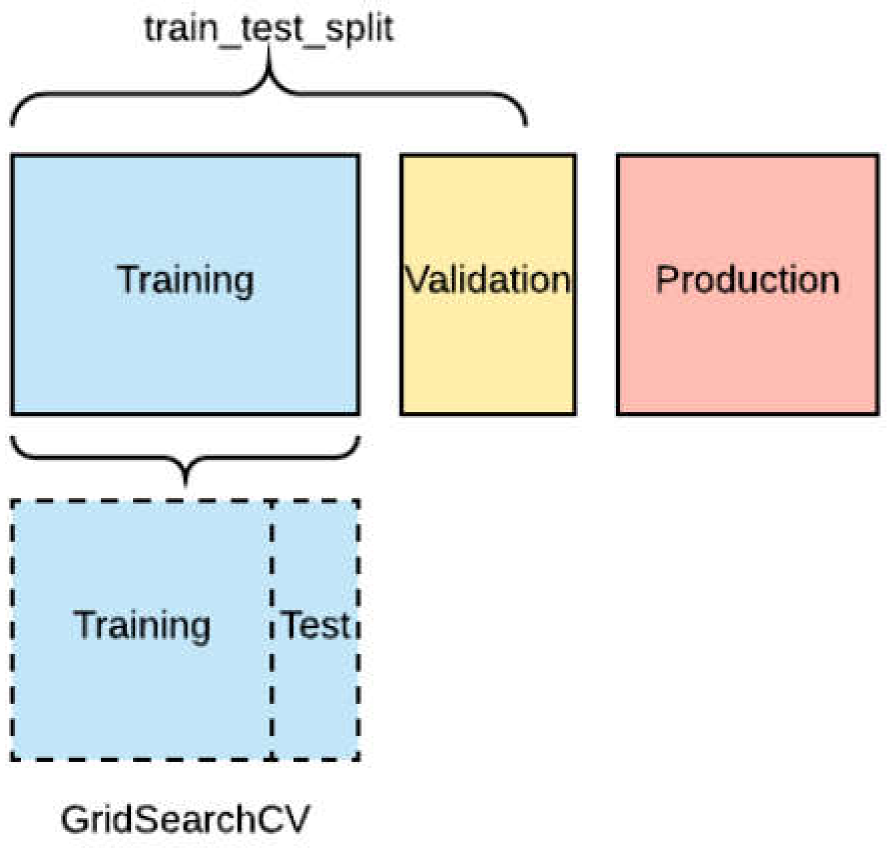
- Training score much higher than test score.
- The standard deviation of the test score is large.

Detecting overfitting

- CV Training Score >> CV Test Score
  - *overfitting in model fitting stage*
  - reduce complexity of classifier
  - get more training data
  - increase cv number
- CV Test Score >> Validation Score
  - *overfitting in model tuning stage*
  - decrease cv number
  - decrease size of parameter grid



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()`.

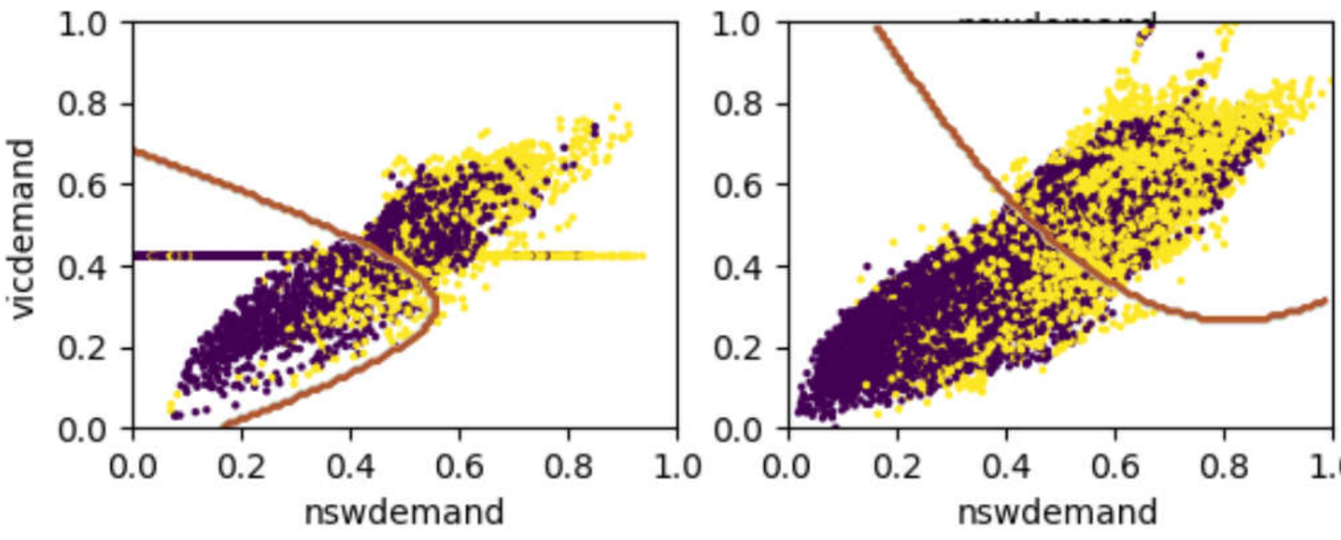
```
# Fit your pipeline using GridSearchCV with three folds
grid_search = GridSearchCV(
  pipe, params, cv=3, return_train_score=True)

# 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

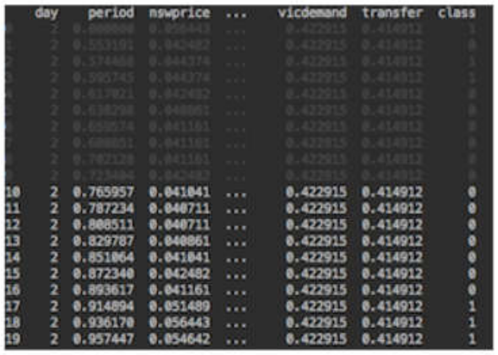


How to protect the model against dataset shit is to Regularly retrain it using only a batch of recent data, a technique known as windowing.

Windows

Sliding window

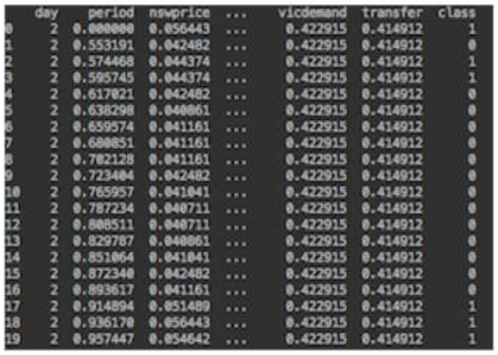
```
window = (t_now-window_size+1):t_now
sliding_window = elec.loc[window]
```



```
    day    period  nswprice  ...   vicdemand  transfer  class
0     2  0.000000  0.056443  ...    0.422915  0.414912      1
1     2  0.553191  0.042482  ...    0.422915  0.414912      0
2     2  0.574468  0.044374  ...    0.422915  0.414912      1
3     2  0.595745  0.044374  ...    0.422915  0.414912      1
4     2  0.617021  0.042482  ...    0.422915  0.414912      0
5     2  0.638298  0.040061  ...    0.422915  0.414912      0
6     2  0.659574  0.041161  ...    0.422915  0.414912      0
7     2  0.680851  0.041161  ...    0.422915  0.414912      0
8     2  0.702128  0.041161  ...    0.422915  0.414912      0
9     2  0.723404  0.042482  ...    0.422915  0.414912      0
10    2  0.765957  0.041041  ...    0.422915  0.414912      0
11    2  0.787234  0.040711  ...    0.422915  0.414912      0
12    2  0.808511  0.040711  ...    0.422915  0.414912      0
13    2  0.829787  0.040061  ...    0.422915  0.414912      0
14    2  0.851064  0.041041  ...    0.422915  0.414912      0
15    2  0.872340  0.042482  ...    0.422915  0.414912      0
16    2  0.893617  0.041161  ...    0.422915  0.414912      0
17    2  0.914894  0.051489  ...    0.422915  0.414912      1
18    2  0.936170  0.056443  ...    0.422915  0.414912      1
19    2  0.957447  0.054642  ...    0.422915  0.414912      1
```

Expanding window

```
window = 0:t_now
expanding_window = elec.loc[window]
```



```
    day    period  nswprice  ...   vicdemand  transfer  class
0     2  0.000000  0.056443  ...    0.422915  0.414912      1
1     2  0.553191  0.042482  ...    0.422915  0.414912      0
2     2  0.574468  0.044374  ...    0.422915  0.414912      1
3     2  0.595745  0.044374  ...    0.422915  0.414912      1
4     2  0.617021  0.042482  ...    0.422915  0.414912      0
5     2  0.638298  0.040061  ...    0.422915  0.414912      0
6     2  0.659574  0.041161  ...    0.422915  0.414912      0
7     2  0.680851  0.041161  ...    0.422915  0.414912      0
8     2  0.702128  0.041161  ...    0.422915  0.414912      0
9     2  0.723404  0.042482  ...    0.422915  0.414912      0
10    2  0.765957  0.041041  ...    0.422915  0.414912      0
11    2  0.787234  0.040711  ...    0.422915  0.414912      0
12    2  0.808511  0.040711  ...    0.422915  0.414912      0
13    2  0.829787  0.040061  ...    0.422915  0.414912      0
14    2  0.851064  0.041041  ...    0.422915  0.414912      0
15    2  0.872340  0.042482  ...    0.422915  0.414912      0
16    2  0.893617  0.041161  ...    0.422915  0.414912      0
17    2  0.914894  0.051489  ...    0.422915  0.414912      1
18    2  0.936170  0.056443  ...    0.422915  0.414912      1
19    2  0.957447  0.054642  ...    0.422915  0.414912      1
```

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

Dataset shift detection

```
# t_now = 40000, window_size = 20000
clf_full = RandomForestClassifier().fit(X, y)
clf_sliding = RandomForestClassifier().fit(sliding_X, sliding_y)
```

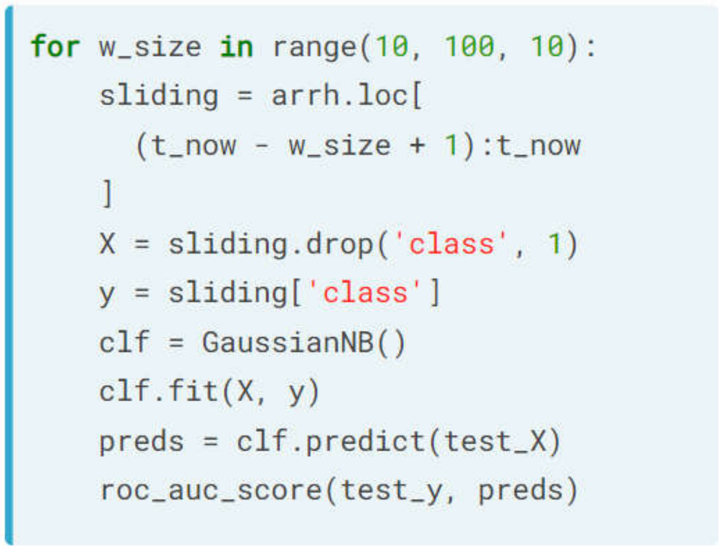
```
# Use future data as test
test = elec.loc[t_now:elec.shape[0]]
test_X = test.drop('class', 1); test_y = test['class']
```

```
roc_auc_score(test_y, clf_full.predict(test_X))
roc_auc_score(test_y, clf_sliding.predict(test_X))
```

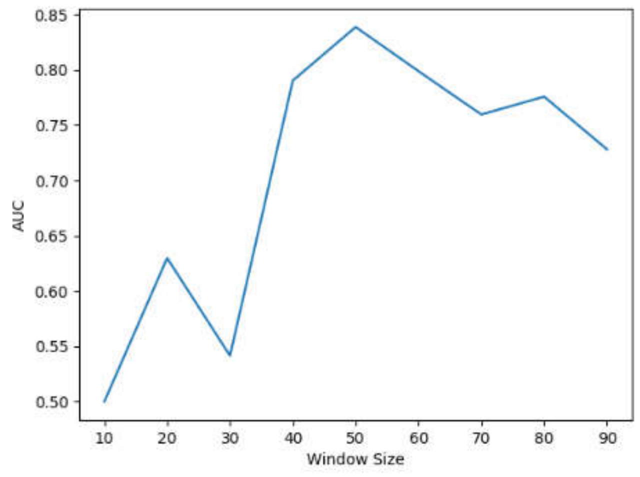
```
0.775
0.780
```

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

Window size



```
for w_size in range(10, 100, 10):
    sliding = arrh.loc[
      (t_now - w_size + 1):t_now
    ]
    X = sliding.drop('class', 1)
    y = sliding['class']
    clf = GaussianNB()
    clf.fit(X, y)
    preds = clf.predict(test_X)
    roc_auc_score(test_y, preds)
```



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`.

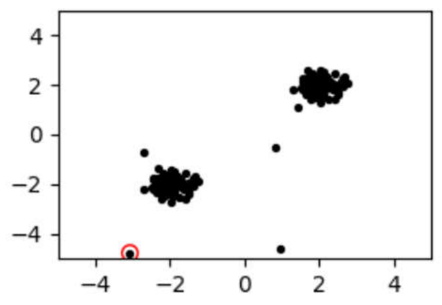
- Create a pipeline with `SelectKBest()` as step `ft` and `RandomForestClassifier()` as step `clf`.
- Create a parameter grid to tune `k` in `SelectKBest()` and `max_depth` in `RandomForestClassifier()`.
- Use `GridSearchCV()` to optimize your pipeline against that grid and data containing only those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

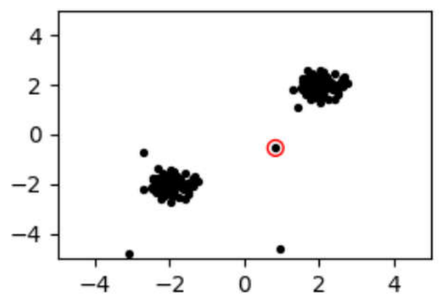
```
# Create a pipeline
pipe = Pipeline([
  ('ft', SelectKBest()), ('clf', RandomForestClassifier(random_state=2))])
# Create a parameter grid
grid = {'ft__k':[5, 10], 'clf__max_depth':[10, 20]}
# Execute grid search CV on a dataset containing under 50s
grid_search = GridSearchCV(pipe, param_grid=grid)
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:

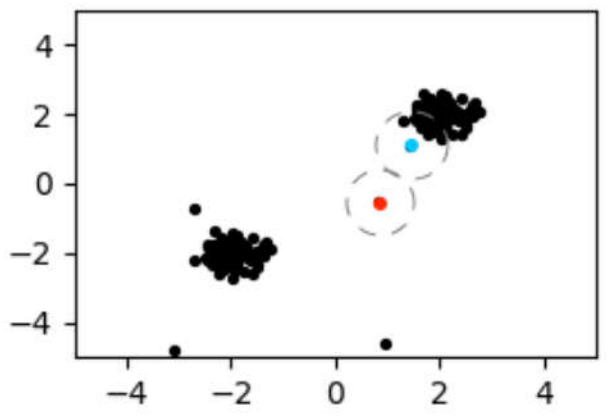
When there is a challenge of modeling data without any, or with very few, labels. This takes you into a journey into anomaly detection, a kind of unsupervised modeling, as well as distance-based learning, where beliefs about what constitutes similarity between two examples can be used in place of labels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

- **Outlier**: a datapoint that lies outside the range of the majority of the data
- **Local outlier**: a datapoint that lies in an isolated region without other data





Local outlier factor (LoF)

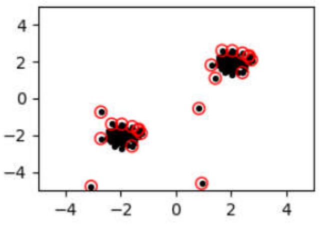


Local outlier factor (LoF)

```
from sklearn.neighbors import
   LocalOutlierFactor as lof
clf = lof()
y_pred = clf.fit_predict(X)
```

```
confusion_matrix(
   y_pred, ground_truth)
```

```
array([[  5,  16],
       [  0, 184]])
```



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

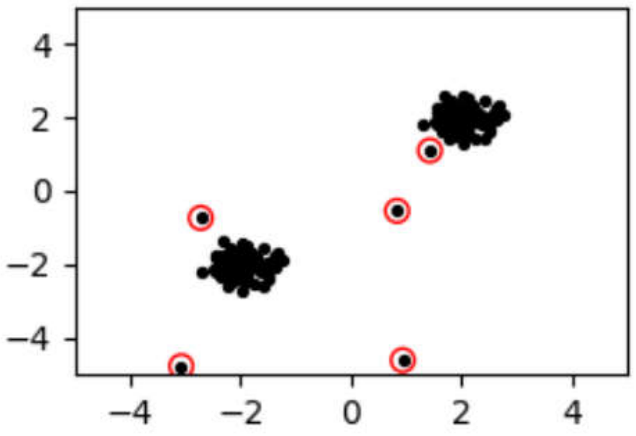
What percent of data is anomalous ? ask user and include it in PARAM as contamination

Local outlier factor (LoF)

```
clf = lof(contamination=0.02)
y_pred = clf.fit_predict(X)
```

```
confusion_matrix(
   y_pred, ground_truth)
```

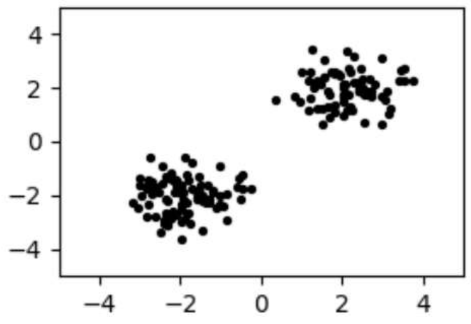
```
array([[  5,   0],
       [  0, 200]])
```



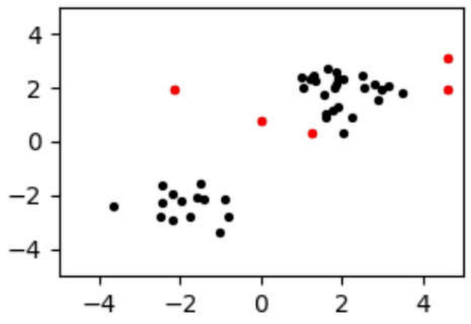
Novelty detection

Detect Anomalies in future data?

Training data without anomalies:



Future / test data with anomalies:



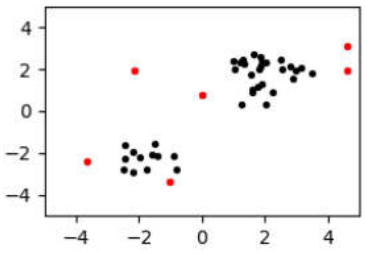
Novelty LoF

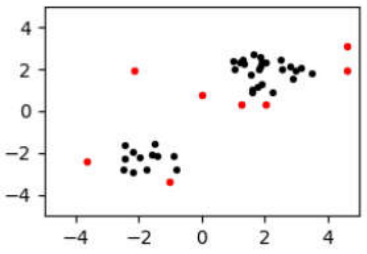
Workaround

```
preds = lof().fit_predict(
    np.concatenate([X_train, X_test]))
preds = preds[X_train.shape[0]:]
```

Novelty LoF

```
clf = lof(novelty=True)
clf.fit(X_train)
y_pred = clf.predict(X_test)
```

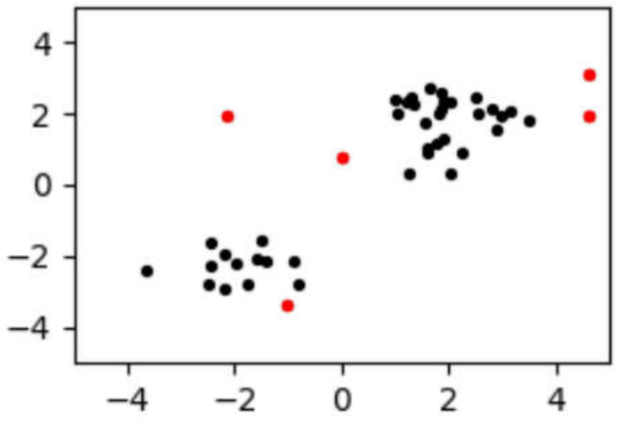




Isolation Forests

```
clf = IsolationForest()
clf.fit(X_train)
y_scores = clf.score_samples(X_test)
```

```
clf = LocalOutlierFactor(novelty=True)
clf.fit(X_train)
y_scores = clf.score_samples(X_test)
```



Novelty detection

```
clf_lof = LocalOutlierFactor(novelty=True).fit(X_train)
clf_isf = IsolationForest().fit(X_train)
clf_svm = OneClassSVM().fit(X_train)
```

```
roc_auc_score(y_test, clf_lof.score_samples(X_test)
```

```
0.9897
```

```
roc_auc_score(y_test, clf_isf.score_samples(X_test))
```

```
0.9692
```

```
roc_auc_score(y_test, clf_svm.score_samples(X_test))
```

```
0.9948
```