Logistic regression

```
from sklearn.linear_model import
LogisticRegression, LogisticRegressionCV
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import Stratifie-
dKFold
from sklearn.preprocessing import PolynomialFe-
atures
from sklearn.model_selection import GridSearchCV
# Create classifier
logit = LogisticRegression(solver='lbfgs',
n_jobs=-1, random_state=7)
# Fit and predict right away...
logit.fit(X_train, y_train)
logit.score(X_test, y_test)
#... or use cross validation for parameter tuning
## solution 1
skf = StratifiedKFold(n_splits=5, shuffle=True,
random_state=17)
c_values = np.logspace(-2, 3, 500)
grid_logit = LogisticRegressionCV(Cs=c_values,
cv=skf, verbose=1, n_jobs=-1)
grid_logit.fit(X_poly, y)
## solution 2
param_grid_logit = {'logisticregression__C':
np.logspace(-5, 0, 6)}
grid_logit = GridSearchCV(logit, param_grid_logit,
return_train_score=True, cv=3, n_jobs=-1)
grid_logit.fit(text_train, y_train)
# Check accuracy and model parameters
logit.score(X_test, y_test)
grid_logit.best_params_, grid_logit.best_score_
# Complexify the model for in case of non-linear
boundaries
poly = PolynomialFeatures(degree=[1-7])
X_poly = poly.fit_transform(X)
# Special case of text processing
## for CV
## for some reason n_jobs > 1 won't work with
GridSearchCV's n_jobs > 1
text_pipe_logit = make_pipeline(CountVectorizer(),
LogisticRegression(solver='lbfgs', n_jobs=1,
random_state=7))
```

Logistic regression (cont)

```
## then classical GridSearchCV with text_pipe-
_logit
```

Decision Trees

```
from sklearn import
DecisionTreeClassifier/Regressor
from sklearn.model_selection import GridSearchCV,
cross_val_score
from sklearn.metrics import accuracy_score
from sklearn.tree import export_graphviz
import pydotplus #pip install pydotplus
# Create classifier
tree = DecisionTreeClassifier(criterion='giny',
splitter='best', max_depth=None, min_sample_leaf=,
random_state = , ...)
# Fit and predict right away...
tree.fit(X_train, y_train)
pred_holdout = tree.predict(X_holdout)
# ... or use cross validation for parameter tuning
tree_params = {'max_depth': range(1,11),
               'max_features': range(4,19)}
tree_grid = GridSearchCV(tree, tree_params, cv=5,
n_jobs=-1, verbose=True)
tree_grid.fit(X_train, y_train)
pred_holdout = tree_grid.predict(X_holdout)
# Check accuracy and model parameters
accuracy_score(y_holdout, pred_holdout)
tree_grid.best_params_, tree_grid.best_score_
# Export Decision Tree as png
tree_str = export_graphviz(tree, feature_names=
[feature_names], filled=True, out_file=None)
graph = pydotplus.graph_from_dot_data(tree_str)
graph.write_png(file)
```

Unsupervised Learning

```
# KMeans
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=k, random_state=1)
kmeans.fit(X)
kmeans.labels_
## elbow method to choose the the number of
clusters
inertia = []
for k in range(1, 8):
    kmeans = KMeans(n_clusters=k, random_state-
=1).fit(X)
    inertia.append(np.sqrt(kmeans.inertia_))
# Accuracy measures
## Needing observations' true labels, scaled
### ARI
ari = metrics.adjusted_rand_score(true_labels,
predicted_labels)
### AMI
ami = metrics.adjusted_mutual_info_score(true_la-
bels, predicted_labels,
                                          -
average_method='arithmetic')
## Needing observations' true labels, not scaled
### Homogenity
h = metrics.homogeneity_score(y, algo.labels_)
### Completeness
c = metrics.completeness_score(y, algo.labels_)
### V-measure
v = metrics.v_measure_score(y, algo.labels_)
##
### Silhouette
s = metrics.silhouette_score(X, algo.labels_)
```

Subsampling, cross-validation, pipelines

k Nearest Neighboors

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV,
cross_val_score
from sklearn.metrics import accuracy_score
# Create classifier
knn = KNeighborsClassifier(n_neighbors=10)
# Scale the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_holdout_scaled = scaler.transform(X_holdout)
# Fit and predict right away...
knn.fit(X_train_scaled, y_train)
knn_pred = knn.predict(X_holdout_scaled)
# ... or use cross validation to tune parameter
knn_pipe = Pipeline([('scaler', StandardScaler()),
('knn', KNeighborsClassifier(n_jobs=-1))])
knn_params = {'knn__n_neighbors': range(1, 10)}
knn_grid = GridSearchCV(knn_pipe, knn_params,
cv=5, n_jobs=-1, verbose=True)
knn_grid.fit(X_train, y_train)
knn_pred = knn_grid.predict(X_holdout_scaled)
# Check accuracy and model parameters
accuracy_score(y_holdout, knn_pred)
knn_grid.best_params_, knn_grid.best_score_
```

Random Forests

```
from sklearn.ensemble import
RandomForestRegressor/Classifier
# if big problems of overfit, import ExtraTreesRe-
gressor/Classifier
from sklearn.model_selection import cross_val-
_score, StratifiedKFold, GridSearchCV
from sklearn.metrics import accuracy_score
# Create classifier
```

Random Forests (cont)

```
rf = RandomForestRegressor(n_estimators=100,
criterion='gini', max_features=, min_samples_-
leaf=, max_depth=, njob=-1, random_state=42,
oob_score=True)
## n_estimators — the number of trees in the
forest.
## max_features: m features chosen in p for each
node. For classification sqrt(d), for regression
d/3.
## min_samples_leaf: minimal number of samples in
a leaf. For min_samples 1, for regression 5.
# Fit and predict right away...
rf.fit(X_train, y_train)
y_test = rf.predict(X_test)
# ... or use cross-validation for parameter tuning
skf = StratifiedKFold(n_splits=5, shuffle=True,
random_state=42)
parameters = {'max_features': [4, 7, 10, 13],
'min_samples_leaf': [1, 3, 5, 7], 'max_depth':
[5,10,15,20]}
rfc = RandomForestClassifier(n_estimators=100,
random_state=42, n_jobs=-1, oob_score=True)
gcv = GridSearchCV(rfc, parameters, n_jobs=-1,
cv=skf, verbose=1)
gcv.fit(X, y)
## for n_estimator, also consider checking test
score as a function of nb of trees [0, 1000]
# Check accuracy and model parameters
results = rfc.score(X_test, y_test)
gcv.best_estimator_, gcv.best_score_
# Determine variable importance
importances = forest.feature_importances_
indices = np.argsort(importances)[::-1]
num_to_plot = 10
## don't forget to link indices to variable's name
bars = plt.bar(range(num_to_plot), importances[-
indices[:num_to_plot]], align="center")
```

A word about bootstrap and bagging