Ensembling Classification Models

Module 2 | Chapter 3 | Notebook 7

The company still wants to reduce the number of employees who quit. In this notebook we'll look for the best classification models and combine them to generate the best predictions. At the end of this lesson you will have learned how to use:

- Grid search with k-Nearest Neighbors, logistic regression and random forest
- Grid search with an ensemble classifier

Preparing optimal classification models

Scenario: You work for an international global logistics company, which wants to limit the number of existing employees who leave the company. You should predict which employees are likely to want to leave the company so that measures can be taken to encourage them to stay.

As usual, we'll start by preparing the data.

In [1]:
```
import pandas as pd
import pickle

#load pipeline
pipeline = pickle.load(open("pipeline.p",'rb'))
col_names = pickle.load(open("col_names.p",'rb'))

#gather data
df_train = pd.read_csv('attrition_train.csv')
df_test = pd.read_csv('attrition_test.csv')

#extract features and target
features_train = df_train.drop('attrition', axis=1)
features_test = df_test.drop('attrition', axis=1)

target_train = df_train.loc[:,'attrition']
target_test = df_test.loc[:,'attrition']

#transform data
features_train = pd.DataFrame(pipeline.transform(features_train), columns=col_names)
features_test = pd.DataFrame(pipeline.transform(features_test), columns=col_names)

# look at raw data
features_train.head()
```

Out[1]:

	pca_years_0	pca_years_1	age	gender	businesstravel	distancefromhome	education	joblevel	ma
0	0.385171	-0.156575	30.0	0.0	1.0	5.0	3.0	2.0	
1	-2.348248	-0.406330	33.0	0.0	1.0	5.0	3.0	1.0	
2	-0.781200	-0.233330	45.0	1.0	1.0	24.0	4.0	1.0	
3	-1.181156	-0.535303	28.0	1.0	1.0	15.0	2.0	1.0	
4	-1.447056	0.019780	30.0	1.0	1.0	1.0	3.0	1.0	

The code for that looks like this:

Column number	Column name	Type	Description
0	`'pca_years_0'`	continuous (`int`)	first principal component of the original columns `'totalworkingyears'`, `'years_atcompany'`, `'years_currentrole'`, `'years_lastpromotion'` and `'years_withmanager'`
1	`'pca_years_1'`	continuous (`int`)	second principal component of the original columns `'totalworkingyears'`, `'years_atcompany'`, `'years_currentrole'`, `'years_lastpromotion'` and `'years_withmanager'`
2	`'attrition'`	categorical	Whether the employee left the company (`1`) or not (`0`)
3	`'age'`	continuous (`int`)	The person's age in years
4	`'gender'`	categorical (nominal, `int`)	Gender: male (`1`) or female (`0`)
5	`'businesstravel'`	categorical (ordinal, `int`)	How often the employee is on a business trip: often (`2`), rarely (`1`) or never (`0`)
6	`'distancefromhome'`	continuous (`int`)	Distance from home address to work address in kilometers
7	`'education'`	categorical (ordinal, `int`)	Level of education: doctorate (`5`), master (`4`), bachelor (`3`), apprenticeship(`2`), Secondary school qualifications (`1`)
8	`'joblevel'`	categorical (ordinal, `int`)	Level of responsibility: Executive (`5`), Manager (`4`), Team leader (`3`), Senior employee (`2`), Junior employee (`1`)
9	`'maritalstatus'`	categorical (nominal, `int`)	Marital status: married (`2`), divorced (`1`), single (`0`)

Column number	Column name	Type	Description
10	`'monthlyincome'`	continuous (`int`)	Gross monthly salary in EUR
11	`'numcompaniesworked'`	continuous (`int`)	The number of enterprises where the employee worked before their current position
12	`'overtime'`	categorically (`int`)	Whether or not they have accumulated overtime in the past year (`1`) or not (`0`)
13	`'percentsalaryhike'`	continuous (`int`)	Salary increase in percent within the last twelve months
14	`'stock option levels'`	categorical (ordinal, `int`)	options on company shares: very many (`4`), many (`3`), few (`2`), very little (`1`), none (`0`)
15	`'trainingtimeslastyear'`	continuous (`int`)	Number of training courses taken in the last 12 months

Each row in `df_train` represents an employee

In this lesson we'll perform several grid searches, which will generate a great deal of warnings in this case. We can ignore these without worrying about them. Run the following code cell so that they are not displayed.

In [2]:
```
from sklearn.exceptions import DataConversionWarning, UndefinedMetricWarning
import warnings
warnings.filterwarnings(action='ignore', category=DataConversionWarning)
warnings.filterwarnings(action='ignore', category=UndefinedMetricWarning)
warnings.filterwarnings(action='ignore', category=DeprecationWarning)
```

You've got to know three different classification approaches so far:

- k-Nearest Neighbors (Module 1 Chapter 2).
- Logistic Regression (Module 2 Chapter 2).
- Decision trees and forests (this chapter)

k-Nearest Neighbors

Which of these approaches performs best in predicting employee attrition? In this lesson we'll look at this question by using grid searches. Let's start with k-Nearest Neighbors.

Create a `Pipeline` called `pipeline_knn`, consisting of two steps:

- Standardization with `StandardScaler` (call this step `'std'`.)
- Classification with `KNeighborsClassifier` (call this step `'knn'`).

In [3]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
```

```
pipeline_knn= Pipeline([('std', StandardScaler ()),('knn', KNeighborsClassifier ())])
```

You defined a grid search in *Grid Search (Module 1, Chapter 2)*. We'll proceed similarly here and create the following hyper parameter settings:

In [4]:
```
import numpy as np
k = np.unique(np.geomspace(1, 500, 15, dtype='int'))  # create 15 values between 1 and

search_space_knn = {'knn__n_neighbors': k,  # use the created values as number of neig
                    'knn__weights': ['uniform', 'distance']}
search_space_knn
```

Out[4]:
```
{'knn__n_neighbors': array([  1,   2,   3,   5,   9,  14,  22,  34,  54,  84, 132, 20
5, 320,
        499]),
 'knn__weights': ['uniform', 'distance']}
```

Now run the grid search and see how good the best model is according to cross validation. Follow these steps:

- Import `GridSearchCV` from `sklearn.model_selection`.
- Select feature matrix (`features_train`) and target vector (`target_train`). Use all the features.
- Instantiate the grid search. Name them `model_knn` and use `pipeline_knn` and `search_space_knn`. You should use the F1 score (`'f1'`) as the `scoring` argument to optimize both the recall and the precision. Use five-fold cross validation (`cv` parameter) to evaluate the hyperparameter settings.
- Carry out the grid search with the training data
- Print the `my_model.best_estimator_` and `my_model.best_score_` attributes of `model_knn`.

In [5]:
```
from sklearn.model_selection import GridSearchCV

model_knn = GridSearchCV(estimator=pipeline_knn,
                  param_grid=search_space_knn,
                  scoring='f1',
                  cv=5)
model_knn.fit(features_train,target_train)
print(model_knn.best_estimator_)
print(model_knn.best_score_)
```

```
Pipeline(steps=[('std', StandardScaler()),
                ('knn', KNeighborsClassifier(n_neighbors=1))])
0.28101947510672154
```

The best model only uses one neighbor (`n_neighbors=1`). The parameter `weights` is not shown since it is set to the default value `'uniform'`. Thus all points in each neighborhood are weighted equally. This results in an F1 score of 28.1%. Let's see if the logistic regression performs better than assigning each new data point the category that its closest neighbor has.

Logistic Regression

Import `LogisticRegression` directly from `sklearn.linear_model`.

In [6]:
```
from sklearn.linear_model import LogisticRegression
```

In *Finding the Best Logistic Regression Model with Grid Search and ROC-AUC (Chapter 2)* we created a `Pipeline` with logistic regression, that allowed regularization like in LASSO regression as well as regularization like in ridge regression. This was achieved by specifying the `solver` parameter in `LogisticRegression` with `'saga'`.

Proceed in a similar way here. Create a `Pipeline` called `pipeline_log`, consisting of two steps:

- Standardization with `StandardScaler`. Call this step `'std'`.
- Classification with `LogisticRegression`. Call this step `'log'`. Use the `solver` parameter as we just described, as just said, and specify `max_iter` as `1e4` to give the algorithm 10,000 attempts. `class_weight` should be set to `'balanced'`. Use `random_state=42` for reproducibilty.

In [7]:
```
pipeline_log = Pipeline([('scaler', StandardScaler()),
                         ('log', LogisticRegression(solver='saga',
                                                    max_iter=1e4,
                                                    class_weight='balanced',
                                                    random_state=42))])
```

Now also define the grid of hyperparameter settings that the grid search should iterate through to find the best settings. Use the settings from *Finding the Best Logistic Regression Model with Grid Search and ROC-AUC (Chapter 2)*.

Two parameters of the logistic regression (the step named `'log'` in `pipeline_log`) should be changed: `penalty` (try `'l1'` and `'l2'`) and `C`. With the regularization parameter `C` you can try 14 values between `0.001` (strong regularization) and `1000` (weak regularization), which creates `np.geomspace()`. `np.geomspace()` has the advantage that the distance between values increases and is not constant.

Name the variable `search_space_log`.

In [8]:
```
C_values = np.geomspace(start=0.001, stop=1000, num=14)

search_space_log = {'log__penalty': ['l1', 'l2'],
                    'log__C': C_values}
```

`search_space_log` should now look like this for you:

```
{'log__penalty': ['l1', 'l2'],
 'log__C': array([1.00000000e-03, 2.89426612e-03, 8.37677640e-03,
2.42446202e-02,
        7.01703829e-02, 2.03091762e-01, 5.87801607e-01, 1.70125428e+00,
        4.92388263e+00, 1.42510267e+01, 4.12462638e+01, 1.19377664e+02,
        3.45510729e+02, 1.00000000e+03])}
```

Now run the grid search and see how good the best model is according to cross validation. Follow these steps:

- Instantiate the grid search. Name it `model_log` and use `pipeline_log` and `search_space_log`. You should use the F1 score (`'f1'`) as the `scoring` argument to optimize both the recall and the precision. Use five-fold cross validation (`cv` parameter) to evaluate the hyperparameter settings.
- Carry out the grid search with the training data
- Print the `my_model.best_estimator_` and `my_model.best_score_` attributes of `model_log`.

In [9]:
```
model_log = GridSearchCV(estimator=pipeline_log,
                         param_grid=search_space_log,
                         scoring='f1',
                         cv=5)

model_log.fit(features_train, target_train)

print(model_log.best_estimator_)
print(model_log.best_score_)
```

```
Pipeline(steps=[('scaler', StandardScaler()),
                ('log',
                 LogisticRegression(C=0.2030917620904737,
                                    class_weight='balanced', max_iter=10000.0,
                                    penalty='l1', random_state=42,
                                    solver='saga'))])
0.4730589275938115
```

This gives us the following results:

Model	f1 score
`model_knn`	28.1%
`model_log`	47.3%

The best model uses quite a lot of regularization (`C=0.20`) like in a LASSO regression (`penalty='l1'`) and results in an F1 score of 47.3%. So the logistic regression model performs much better than k-Nearest Neighbors.

We'll try random forest as the third algorithm.

The random forest

Import `RandomForestClassifier` directly from `sklearn.ensemble`.

In [10]:
```
from sklearn.ensemble import RandomForestClassifier
```

Since decision trees do not need standardization, we don't need to use a `Pipeline` for this. Instead, we want to try out these parameter settings during the grid search:

In [11]:
```
search_space_rf = {'max_depth': np.geomspace(start=3, stop=50, num=10, dtype='int'),
                   'min_samples_leaf': np.geomspace(start=1, stop=500, num=10, dtype='
```

Now run the grid search and see how good the best model is according to cross validation. Follow these steps:

- Instantiate the grid search. Call it `model_rf` and use `RandomForestClassifier()` (with the settings `class_weight='balanced'`, `n_estimators = 50` and `random_state=42`) and `search_space_rf`. You should use the F1 score (`'f1'`) as the `scoring` argument. Use five-fold cross validation (`cv` parameter) to evaluate the hyperparameter settings.
- Carry out the grid search on the training data
- Print the `my_model.best_estimator_` and `my_model.best_score_` attributes of `model_rf`.

In [12]:
```
model_rf = GridSearchCV(estimator=RandomForestClassifier(class_weight='balanced',
                                                         n_estimators=50,
                                                         random_state=42),
                        param_grid=search_space_rf,
                        scoring='f1',
                        cv=5)

model_rf.fit(features_train, target_train)

print(model_rf.best_estimator_)
print(model_rf.best_score_)
```

```
RandomForestClassifier(class_weight='balanced', max_depth=7, min_samples_leaf=7,
                       n_estimators=50, random_state=42)
0.49027633482249566
```

The best model uses seven decision levels (`max_depth=7`) with at least seven data points in each leaf (`min_samples_leaf=7`). The resulting model quality of approx. 49% according to the F1 score is the best value so far. Your value may be slightly different because of the random elements in the *random forest*.

In summary, the performance metrics of the models look like this:

Model	f1 score
`model_knn`	28.1%
`model_log`	47.3%
`model_rf`	49.0%

Congratulations: You have applied all your knowledge of classification models to find the best one. According to cross-validation, a random forest is the best model so far with an F1 score of 49%. However, instead of choosing between the classification models, you could combine them. Let's look at that next.

Combining classification models optimally

In *From Decision Trees to Random Forests with Ensembling* we learned how to combine the predictions of classification algorithms into a meta-classifier. `VotingClassifier` from `sklearn.ensemble` implements ensembling very elegantly, without the hassle we had in *From Decision Trees to Random Forests with Ensembling*.

Import `VotingClassifier` directly from `sklearn.ensemble`.

In [13]:
```
from sklearn.ensemble import VotingClassifier
```

`VotingClassifier` allows different ways to combine the predictions of the models. The `voting` parameter controls whether to combine the predicted categories (`voting='hard'`) or the predicted category probabilities (`voting='soft'`). Remember that a *random forest* does the latter. We can try both approaches in the grid search.

You can also use the `weights` parameter to specify whether all the models should contribute equally to the final vote (`weights=None`) or whether they should be weighted (`weights=[weight_model_1, weight_model_2, weight_model_3]`). But which weighting should we use for the latter? The F1 scores that we obtained through cross validation are suitable for this. This would give a model with better performance more influence than a model with worse performance.

So define a new variable called `search_space_ens` . This should be a `dict` . This `dict` has two *keys*: `'voting'` and `'weights'` . The corresponding *values* are always lists. For `'voting'` it has the entries `'soft'` and `'hard'` . For `'weights'` it looks like this: `[None, [model_knn.best_score_, model_log.best_score_, model_rf.best_score_]]` .

In [14]:
```
search_space_ens = {'voting':['soft','hard'],
                    'weights':[None, [model_knn.best_score_, model_log.best_score_, mc
```

Now instantiate the `VotingClassifier` , which we will use for the grid search. In this case, only specify the `estimators` parameter. This takes a `list` of tuples. The tuples consist of names and models, just like the steps in a `Pipeline` . Use the names `'knn'` , `'log'` and `'rf'` for your trained models `model_knn` , `model_log` and `model_rf` . Name your instance `voting_knn_log_rf` .

In [15]:
```
voting_knn_log_rf = VotingClassifier(estimators=[('knn', model_knn), ('log', model_log
```

Now run the grid search and see how good the best meta-model is according to cross validation. Follow these steps:

- Instantiate the grid search. Name it `model_ens` and assign `voting_knn_log_rf` to the `estimator` parameter.
  In addition, assign `search_space_ens` to `param_grid` . Use the F1 score as the

`scoring` parameter again. In order to keep the computing requirements small, we'll only use triple cross-validation this time.

- Carry out the grid search with the training data.
- Print the `my_model.best_estimator_`, `my_model.best_score_` and `my_model.best_params_` attributes of `model_ens`.

Important: `model_ens` is very complex and will therefore take a very long time to calculate (we estimate at least 5 minutes). Therefore **definitely** use the parameter `n_jobs=-1` when instantiating `GridSearchCV`! This distributes the computing load over all available CPU cores, which significantly speeds up the search.

In [16]:
```
model_ens = GridSearchCV(estimator=voting_knn_log_rf,
                         param_grid=search_space_ens,
                         scoring='f1',
                         cv=3,
                         n_jobs=-1)

model_ens.fit(features_train, target_train)

print(model_ens.best_estimator_)
print(model_ens.best_score_)
print(model_ens.best_params_)
```

```
VotingClassifier(estimators=[('knn',
                              GridSearchCV(cv=5,
                                           estimator=Pipeline(steps=[('std',
                                                                      StandardScaler
()),
                                                                     ('knn',
                                                                      KNeighborsClass
ifier())]),
                                           param_grid={'knn__n_neighbors': array([
1,   2,   3,   5,   9,  14,  22,  34,  54,  84, 132, 205, 320,
       499]),
                                                       'knn__weights': ['uniform',
                                                                        'distance']},
                                           scoring='f1')),
                             ('log',
                              GridSearchCV(cv=5,
                                           estimator=Pipeline(steps=[('scaler',
                                                                      StandardScaler
()...
       4.92388263e+00, 1.42510267e+01, 4.12462638e+01, 1.19377664e+02,
       3.45510729e+02, 1.00000000e+03]),
                                                       'log__penalty': ['l1',
                                                                        'l2']},
                                           scoring='f1')),
                             ('rf',
                              GridSearchCV(cv=5,
                                           estimator=RandomForestClassifier(class_wei
ght='balanced',
                                                                            n_estimat
ors=50,
                                                                            random_st
ate=42),
                                           param_grid={'max_depth': array([ 3,  4,
5,  7, 10, 14, 19, 26, 36, 49]),
                                                       'min_samples_leaf': array([
1,   1,   3,   7,  15,  31,  62, 125, 250, 499])},
                                           scoring='f1'))])
0.49335232668566004
{'voting': 'hard', 'weights': None}
```

The best meta-model uses the categorical predictions (`voting='hard'`) and weights all models equally (`weights=None`). The resulting model quality of 49.3% according to the F1 score is about as high as with logistic regression and the random forest.

In summary, the performance metrics of the models look like this:

Model	f1 score
`model_knn`	28.1%
`model_log`	47.3%
`model_rf`	49.0%
`model_ens`	49.3%

Congratulations: You have used ensembling to combine machine learning models with each other and therefore slightly improve the *f1 score*. At the same time, you have also learned how

to optimize this combination. A grid search came in handy again. Finally, we'll evaluate the classification models using the test data.

Evaluating the best classification models

Once you've gained an overview of which models are promising and how well they perform approximately using the grid search, you should evaluate them again with new test data. We can now use three model quality measures for this evaluation to examine our models in detail:

- Precision
- Recall
- F1 score

Now import the functions that belong to the model quality metrics directly from `sklearn.metrics`:

- `precision_score()`
- `recall_score()`
- `f1_score()`

Important: If you set `voting='hard'`, the `VotingClassifier` doesn't have `my_classifier.predict_proba()`, so no probabilities are generated for the classification. So it makes no sense to use `roc_auc_score()` in this case. So we won't use that here.

In [17]:
```
from sklearn.metrics import precision_score, recall_score, f1_score
```

Write a `for` loop to iterate through the three promising models `model_log`, `model_rf` and `model_ens` and for each model:

- make predictions based on the test data
- calculate and print the model quality metrics.

In [18]:
```
for clf in [model_log, model_rf, model_ens]:
    target_test_pred = clf.predict(features_test)

    print('\nPrecision: ', precision_score(target_test, target_test_pred))
    print('Recall: ', recall_score(target_test, target_test_pred))
    print('F1: ', f1_score(target_test, target_test_pred))
```

```
Precision:  0.34615384615384615
Recall:  0.7714285714285715
F1:  0.47787610619469023

Precision:  0.45161290322580644
Recall:  0.4
F1:  0.4242424242424243

Precision:  0.4852941176470588
Recall:  0.4714285714285714
F1:  0.4782608695652174
```

We received the following values (yours may vary slightly):

Model	*precision*	*recall*	*f1 score*
`model_log`	34.6%	77.1%	47.8%
`model_rf`	45.2%	40.0%	42.4%
`model_ens`	48.5%	47.1%	47.8%

For us the meta-model (`model_ens`) has the most balanced performance. It can identify about half of the people who leave the company (recall) and generate predictions of `'attrition'` cases, about half of which are true (precision). The other two models are either better at identifying the people leaving the company (high recall value for `model_log`) or better at classifying people who actually leave the company as such (higher precision value for `model_rf').

So what should we do now? We would recommend the logistic regression model to the logistics company for the following reasons:

- If you want to identify people leaving the company, it is more important to find those people (high recall) than to identify only those people in the predictions who will actually leave (high precision). `model_log` has the highest *recall* value. You can assume that it will find out about 77% of the people who will leave the company. But if you approach all the people this model identifies as `'attrition'` cases, you have to assume that 65% of them don't actually want to leave the company after all (precision of only about 35%).
- If the company doesn't agree with this trade-off between recall and precision, `model_log` offers the best option for a flexible shift in the decision threshold of when to approach people.

If the company wants to find out why employees are leaving the company, you could train a decision tree and examine its decision rules, for example. We'll look at how to visualize and interpret these in Module 3.

Congratulations: You have combined classification models with each other and created a very balanced meta-model. Nevertheless, we would recommend that the company uses a logistic regression model in this case.

The final data pipeline

In order to be able to pass on your model to the development department, you should now merge all the steps into a final data pipeline. Now all of the *preprocessing* stage takes place in `pipeline` . Now all that's missing is to add the trained model (we decided to use logistic regression in `model_log` , of course you can try out other models). To do this, create a new pipeline `pred_pipe` which combines `pipeline` and `model_log` .

In [19]:
```
pred_pipe = Pipeline([('preprocessing', pipeline), ('prediction', model_log)])
```

Now you can test your pipeline. We've made *attrition_aim.csv* available to you for this. The file contains the employee data from a larger department and you should predict how many employees are at risk of leaving. Import *attrition_aim.csv* and store the `DataFrame` in the variable `features_aim`, then pass it to `my_estimator.predict()` method of `pred_pipe`.

How many employees are predicted to leave?

In [20]:
```
features_aim = pd.read_csv('attrition_aim.csv')
print(sum(pred_pipe.predict(features_aim)), len(features_aim))
```

```
19 50
```

We have identified 19 out of 50 employees in the target data set who might want to leave, so according to the model the company definitely needs to act!

Congratulations: You have prepared a model to the point where it can be easily embedded in a production environment. The company embraces your pipeline and thanks you very much for your hard work.

Remember:

- Ensembling with `sklearn.ensemble.VotingClassifier`
- `VotingClassifier` expects a list of `tuple` pairs of name and models for `estimators`.
- Meta models can also be optimized with a grid search.

Do you have any questions about this exercise? Look in the forum to see if they have already been discussed.

Found a mistake? Contact Support at support@stackfuel.com.
