04-fm-test

May 16, 2018

In [1]:
```
# This is an initialization cell that is not part of the presentation.
# This cell should be run while not in the rise mode

import numpy as np

np.random.seed(0) # make notebook deterministic

from sklearn.decomposition import TruncatedSVD # import model that we will use but do no

# Disable warnings during the presentation
import warnings
warnings.filterwarnings("ignore")


#CSS customization
from IPython.core.display import HTML, display

# set width cell to screen width
display(HTML("<style>.container { width:100% !important; }</style>"))

# default font-size is 10pt
# anyway the code below set font size for code cells
HTML("""<style>
.CodeMirror pre {
    font-size: 11pt;
}
</style>""")

# increase font size of pd.DataFrame
HTML("""<style>
     table.dataframe {
font-size:24px;}
</style>""")

# to increase font size in a markdown cell
#<font size=5px>test</font>
```

```
<IPython.core.display.HTML object>
```

```
Out[1]: <IPython.core.display.HTML object>
```

1 Pipelines and Gridsearch with scikit-learn

May 16 2018

1.1 Florent Martin

1.2 Koen van Woerden

2 Goal of the talk:

- **Pipeline**
- **Gridsearch**
- **Scikit-learn**

3 What is a data pipeline?

4 What is a data pipeline?

Data	Liquid

5 Pipeline $\Rightarrow$ easily experiment

6 What is Gridsearch?

7 Gridsearch Example

Goal: find the best hyperparameter for logistic regression among
Regularization type: L1, L2
$C = 0.1, 1, 10, 100$

8 Pipeline + Gridsearch $\Rightarrow$ scikit-learn to the rescue

1. **Classifying authors**
   - Dataset
   - Baseline model
2. **Pipeline**
   - Build your first pipeline

- Add new transformations
- Add non-scikit-learn transformations
- Keep experimenting

3. **Gridsearch**
   - Hyperparameters
   - With scikit-learn transformations
   - With non-scikit-learn transformations

9 Part 1: Classifying authors

```
<tr>
    <td>INPUT</td> <td></td><td>OUTPUT</td>
</tr>
<tr>
    <td>sentence</td>                                          <td>$\Rightarrow$</td>  <td>autho
</tr>
<tr>
    <td>'Even the very lights from the city bewilder him.' </td><td>$\Rightarrow$</td>  <td>Edga
</tr>
<tr>
    <td>X</td>                                                 <td>$\Rightarrow$</td>  <td>y</td
</tr>
```

```
In [2]: import numpy as np
        import pandas as pd
```

```
In [3]: data = pd.read_csv('../data/talk/data.csv')
```

```
In [4]: data.shape
```

```
Out[4]: (14684, 2)
```

```
In [5]: data.sample(n=5)
```

```
Out[5]:                                                    text     author
        10115  "Our first slide into the abyss itself, from t...        Poe
        5906   He heard my account of the self dissolution of...    Shelley
        5777   Nor has he yet had any difficulty in obtaining...  Lovecraft
        8462   We examined, first, the furniture of each apar...        Poe
        1457   I did this at some little risk, and before clo...        Poe
```

```
In [6]: X, author = data['text'], data['author']
```

```
In [7]: author.value_counts()
```

```
Out[7]: Poe          5963
        Shelley      4465
        Lovecraft    4256
        Name: author, dtype: int64
```

Poe
|
Shelley
|
Lovecraft
---|---|--- | |

10 scikit-learn basics

- Objects have `fit` method
- Objects have `transform` or `predict` method

```
In [ ]: model.fit(X, y)
        model.transform(X)

In [ ]: model.fit_transform(X, y)

In [ ]: model.fit(X, y)
        model.predict(X)
```

11 Turn labels into integers

```
In [8]: from sklearn.preprocessing import LabelEncoder

In [9]: label_encoder = LabelEncoder()

In [10]: label_encoder.fit(author);

In [11]: y = label_encoder.transform(author)

In [12]: pd.DataFrame({'author': author[:5], 'y': y[:5]})
```

Out[12]:

	author	y
0	Shelley	2
1	Lovecraft	0
2	Shelley	2
3	Poe	1
4	Poe	1

11.1 Bag of words: convert strings to vectors (one-hot encoding)

12 Baseline model: Bag of Words + Logistic Regression

```
In [13]: from sklearn.feature_extraction.text import CountVectorizer

In [14]: cvec = CountVectorizer()

In [15]: cvec.fit(X);
```

```
In [16]: X_cvec = cvec.transform(X)

In [17]: type(X_cvec)

Out[17]: scipy.sparse.csr.csr_matrix

In [18]: X_cvec.shape

Out[18]: (14684, 22476)
```

13 Logistic regression

```
In [19]: from sklearn.linear_model import LogisticRegression

In [20]: logistic_regression = LogisticRegression()

In [21]: logistic_regression.fit(X_cvec, y)

Out[21]: LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                            intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                            penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
                            verbose=0, warm_start=False)
```

13.0.1 (Multi-class: One-versus-Rest)

14 Predict author of sentence

```
In [22]: rand_sentence = X.sample()

In [23]: print(rand_sentence.iloc[0])

I always attributed my failure at these points to the disordered state of his health.


In [24]: rand_sentence_vec = cvec.transform(rand_sentence)

In [25]: logistic_regression.predict_proba(rand_sentence_vec)

Out[25]: array([[0.04726737, 0.85903555, 0.09369708]])

In [26]: label_encoder.classes_

Out[26]: array(['Lovecraft', 'Poe', 'Shelley'], dtype=object)

In [27]: idx = rand_sentence.index[0]
         author.loc[idx]

Out[27]: 'Poe'
```

15 Accuracy

```
In [28]: logistic_regression.score(X_cvec, y)
```

```
Out[28]: 0.9741214927812585
```

16 Generalization: try model on new data

```
In [29]: val = pd.read_csv('../data/talk/val.csv')
```

```
In [30]: val.shape
```

```
Out[30]: (4895, 2)
```

```
In [31]: X_val, author_val  = val['text'], val['author']
```

```
In [32]: X_val_cvec = cvec.transform(X_val)
```

```
In [33]: y_val = label_encoder.transform(author_val)
```

```
In [34]: logistic_regression.score(X_val_cvec, y_val)
```

```
Out[34]: 0.8175689479060265
```

16.1 Repetitive code

16.2 Solution: pipeline

17 Part 2: Pipeline

18 Combine all transformations in a Pipeline

```
In [50]: cvec = CountVectorizer()
         logistic_regression = LogisticRegression()
```

```
In [51]: X_cvec = cvec.fit_transform(X)
```

```
In [44]: X_svd = svd.fit_transform(X_cvec)
```

```
In [45]: logistic_regression.fit(X_svd, y);
```

```
In [46]: logistic_regression.score(X_svd, y)
```

```
Out[46]: 0.4443612094797058
```

```
In [47]: X_val_cvec = cvec.transform(X_val)
```

```
In [48]: X_val_svd = svd.transform(X_val_cvec)
```

```
In [49]: logistic_regression.score(X_val_svd, y_val)
```

```
Out[49]: 0.44473953013278855

In [ ]: cvec = CountVectorizer()
        svd = TruncatedSVD()
        logistic_regression = LogisticRegression()

In [ ]: X_cvec = cvec.fit_transform(X)

In [ ]: X_svd = svd.fit_transform(X_cvec)

In [ ]: logistic_regression.fit(X_svd, y);

In [ ]: logistic_regression.score(X_svd, y)

In [ ]: X_val_cvec = cvec.transform(X_val)

In [ ]: X_val_svd = svd.transform(X_val_cvec)

In [ ]: logistic_regression.score(X_val_svd, y_val)
```

- Many **intermediate variables**
- Transformations spread out over the notebook
- **Experimenting** is **difficult**
- **Solution**: create a **Pipeline object**

```
In [52]: from sklearn.pipeline import Pipeline

In [61]: pipeline = Pipeline(steps= [   ('cvec'   , CountVectorizer()),
                                        ('logreg', LogisticRegression())  ]  )

In [62]: pipeline.fit(X, y);

In [63]: pipeline.score(X, y)

Out[63]: 0.9741214927812585

In [64]: pipeline.score(X_val, y_val)

Out[64]: 0.8175689479060265

In [77]: rand_sentence = X_val.sample()

In [78]: print(rand_sentence.iloc[0])

We all observed the visitation of these feelings, and none regretted them so much as Perdita.


In [79]: pipeline.predict_proba(rand_sentence)
```

```
Out[79]: array([[0.00159868, 0.05449921, 0.94390211]])

In [80]: pipeline.predict(rand_sentence)

Out[80]: array([2])

In [81]: label_encoder.classes_

Out[81]: array(['Lovecraft', 'Poe', 'Shelley'], dtype=object)

In [82]: author[rand_sentence.index]

Out[82]: 4132    Shelley
         Name: author, dtype: object
```

18.1 Under the hood of Pipeline

```
In [ ]: pipeline = Pipeline(steps=  [ ('first_transformation', first_transformation),
                                        ...
                                      ('last_transformation', last_transformation)          ] )

In [ ]: pipeline.fit(X,y)
```

18.2 Scikit-learn does

```
In [ ]: X_first = first_transformation.fit_transform(X)
        X_second = second_transformation.fit_transform(X_first)
        ...
        X_last = last_transformation.fit(X_previous_last)
```

- All step but the last *must* implement a `fit` and `transform` method
- The last step *must* implement a `fit` method, and a `transform` or `predict` method as well

19 Add a non-scikit-learn transformation to the Pipeline

19.1 Lemmatizer

lemma = **dictionary entry**
swimming, **swims**, **swim** $\Rightarrow$ same **lemma** $\Rightarrow$ **swim**
Lemmatizer: word $\mapsto$ lemma

19.2 nltk = natural language toolkit (NLP library)

```
In [83]: from nltk.stem import WordNetLemmatizer

In [84]: class Lemmatizer():
             def fit(self, X, y=None):
                 return self

             def transform(self, X, y=None):
```

```
                lem = WordNetLemmatizer()
                lower = X.str.lower()
                tokenized = lower.str.split(' ')
                lemmatized = tokenized.apply(lambda l: " ".join([lem.lemmatize(word) for word i
                return lemmatized
```

```
In [85]: lemmatizer = Lemmatizer()
```

```
In [86]: sentence = pd.Series(data=['Cows and pigs are common animals on farms'])
```

```
In [87]: lemmatizer.transform(sentence).iloc[0]
```

```
Out[87]: 'cow and pig are common animal on farm'
```

```
In [88]: pipeline = Pipeline(steps=[  ('lem', Lemmatizer()),
                                      ('cvec', CountVectorizer()),
                                      ('logreg', LogisticRegression())   ])
```

```
In [89]: pipeline.fit(X, y);
```

```
In [90]: pipeline.score(X, y)
```

```
Out[90]: 0.9720784527376737
```

```
In [91]: pipeline.score(X_val, y_val)
```

```
Out[91]: 0.8145045965270684
```

20 Adding Gensim word2vec

```
In [92]: from gensim.models.word2vec import Word2Vec
```

```
In [93]: class GensimWord2Vec():
             def fit(self, X, y=None):
                 self.model = Word2Vec(X)
                 return self

             def transform(self, X, y=None):
                 lower = X.str.lower()
                 tokenized = lower.str.split(' ')
                 vectors = tokenized.apply(lambda l: [self.model[word] for word in l if word in
                 def average(l):
                     if l == []:
                         return np.zeros(self.model.vector_size)
                     else:
                         return np.mean(l, axis=0)
                 vectors = vectors.apply(average)
                 vectors = vectors.apply(pd.Series)
                 return vectors
```

```
In [94]: pipeline = Pipeline(steps=  [   ('word2vec', GensimWord2Vec()),
                                         ('logreg', LogisticRegression())    ]  )

In [95]: pipeline.fit(X, y);

In [96]: pipeline.score(X, y)

Out[96]: 0.41691637156088257

In [97]: pipeline.score(X_val, y_val)

Out[97]: 0.4063329928498468
```

20.1 Feature unions: Combine bag of words and word2vec

```
In [98]: from sklearn.pipeline import FeatureUnion

In [99]: lem_cvec = Pipeline(steps = [('lem', Lemmatizer()),
                                       ('cvec', CountVectorizer())])

In [100]: feature_union = FeatureUnion([('lem_cvec', lem_cvec),
                                        ('gensimw2v', GensimWord2Vec())])

In [101]: pipeline = Pipeline( [  ('feature_union', feature_union),
                                  ('logreg', LogisticRegression())                    ])

In [102]: pipeline.fit(X, y);

In [103]: pipeline.score(X, y)

Out[103]: 0.9732361754290384

In [104]: pipeline.score(X_val, y_val)

Out[104]: 0.8175689479060265
```

21 Further experiment: tf-idf

```
In [105]: from sklearn.feature_extraction.text import TfidfVectorizer

In [106]: pipeline = Pipeline(steps=  [  ('lem', Lemmatizer()),
                                         ('tfidf', TfidfVectorizer()),
                                         ('logreg', LogisticRegression())    ])

In [107]: pipeline.fit(X, y);

In [108]: pipeline.score(X, y)

Out[108]: 0.8962816671206756

In [109]: pipeline.score(X_val, y_val)

Out[109]: 0.802655771195097
```

22 Further experiment: Naive Bayes classifier

```
In [110]: from sklearn.naive_bayes import MultinomialNB
```

```
In [111]: pipeline = Pipeline(steps=  [    ('CountVectorizer', CountVectorizer()),
                                           ('NaiveBayes', MultinomialNB())            ])
```

```
In [112]: pipeline.fit(X, y);
```

```
In [113]: pipeline.score(X, y)
```

```
Out[113]: 0.9156224461999455
```

```
In [114]: pipeline.score(X_val, y_val)
```

```
Out[114]: 0.8330949948927477
```

23 Part 3: Gridsearch

23.1 What is a hyperparameter?

23.2 Examples

- learning rate
- regularization coefficient
- number of hidden layers in a neural network
- ...

23.3 Responsibility of the data scientist

change **hyperparameter** ⇒ change **model** ⇒ change **performance**

24 Baseline model

```
In [115]: cvec = CountVectorizer()
```

```
In [116]: X_cvec = cvec.fit_transform(X)
```

```
In [117]: logistic_regression = LogisticRegression(C=1)
```

- `C` is a **hyperparameter**
- How do we know about `C`?

```
In [120]: LogisticRegression?
```

```
In [118]: logistic_regression = LogisticRegression(C=1)
          logistic_regression.fit(X_cvec, y);
          logistic_regression.score(X_cvec, y)
```

```
Out[118]: 0.9741214927812585
```

```
In [119]: X_val_cvec = cvec.transform(X_val)
          logistic_regression.score(X_val_cvec, y_val)
```

```
Out[119]: 0.8175689479060265
```

25 Gridsearch

- Our previous model depends on a **hyperparameter** `C`
- Changing `C` changes the performance ⇒ **Try** different `C`
- Keep track of the results! (Who remembers the results we got?)
- We want this to be done automatically
- Gridsearch is what we need

26 Gridsearch in scikit-learn

```
In [121]: from sklearn.model_selection import GridSearchCV

In [122]: gridsearch = GridSearchCV(estimator=LogisticRegression(),
                                    param_grid={'C': [0.1, 1, 10, 100, 1000]}, verbose=3)

In [123]: gridsearch.fit(X_cvec, y)

Fitting 3 folds for each of 5 candidates, totalling 15 fits
[CV] C=0.1 ...
[CV] ... C=0.1, score=0.7757352941176471, total=   0.4s
[CV] C=0.1 ...


[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.4s remaining:    0.0s


[CV] ... C=0.1, score=0.7662921348314606, total=   0.4s
[CV] C=0.1 ...


[Parallel(n_jobs=1)]: Done   2 out of   2 | elapsed:    0.8s remaining:    0.0s


[CV] ... C=0.1, score=0.7741677177600654, total=   0.5s
[CV] C=1 ...
[CV] ... C=1, score=0.8112745098039216, total=   0.7s
[CV] C=1 ...
[CV] ... C=1, score=0.8032686414708886, total=   0.9s
[CV] C=1 ...
[CV] ... C=1, score=0.8078888207643572, total=   0.8s
[CV] C=10 ...
[CV] ... C=10, score=0.7998366013071896, total=   1.2s
[CV] C=10 ...
[CV] ... C=10, score=0.79244126659857, total=   1.5s
[CV] C=10 ...
```

```
[CV] ... C=10, score=0.7968526466380543, total=   1.6s
[CV] C=100 ...
[CV] ... C=100, score=0.7792075163398693, total=   1.4s
[CV] C=100 ...
[CV] ... C=100, score=0.780388151174668, total=   1.9s
[CV] C=100 ...
[CV] ... C=100, score=0.7819333742080523, total=   1.6s
[CV] C=1000 ...
[CV] ... C=1000, score=0.769812091503268, total=   1.8s
[CV] C=1000 ...
[CV] ... C=1000, score=0.7671092951991828, total=   1.6s
[CV] C=1000 ...
[CV] ... C=1000, score=0.768444716942571, total=   1.7s


[Parallel(n_jobs=1)]: Done  15 out of  15 | elapsed:   17.9s finished


Out[123]: GridSearchCV(cv=None, error_score='raise',
                estimator=LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercep
                   intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                   penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
                   verbose=0, warm_start=False),
                fit_params=None, iid=True, n_jobs=1,
                param_grid={'C': [0.1, 1, 10, 100, 1000]}, pre_dispatch='2*n_jobs',
                refit=True, return_train_score='warn', scoring=None, verbose=3)

In [124]: gridsearch.best_params_

Out[124]: {'C': 1}

In [125]: gridsearch.best_score_

Out[125]: 0.8074775265595205

In [126]: gridsearch.cv_results_

Out[126]: {'mean_fit_time': array([0.42627247, 0.78037278, 1.46709331, 1.59198896, 1.68246086]),
           'mean_score_time': array([0.00176358, 0.00143449, 0.00157404, 0.00137695, 0.0015467 ]
           'mean_test_score': array([0.77206483, 0.80747753, 0.79637701, 0.7805094 , 0.76845546]
           'mean_train_score': array([0.87806463, 0.97950149, 0.99819533, 0.99979571, 1.
           'param_C': masked_array(data=[0.1, 1, 10, 100, 1000],
                        mask=[False, False, False, False, False],
                  fill_value='?',
                       dtype=object),
           'params': [{'C': 0.1}, {'C': 1}, {'C': 10}, {'C': 100}, {'C': 1000}],
           'rank_test_score': array([4, 1, 2, 3, 5], dtype=int32),
           'split0_test_score': array([0.77573529, 0.81127451, 0.7998366 , 0.77920752, 0.7698120
           'split0_train_score': array([0.87832039, 0.97987331, 0.99856968, 0.99989783, 1.
```

```
 'split1_test_score': array([0.76629213, 0.80326864, 0.79244127, 0.78038815, 0.7671093
 'split1_train_score': array([0.87874144, 0.97875166, 0.99785473, 0.99979569, 1.
 'split2_test_score': array([0.77416718, 0.80788882, 0.79685265, 0.78193337, 0.7684447
 'split2_train_score': array([0.87713206, 0.97987948, 0.99816158, 0.9996936 , 1.
 'std_fit_time': array([0.02773562, 0.10812919, 0.17687165, 0.21361441, 0.08553933]),
 'std_score_time': array([0.0001857 , 0.0001024 , 0.0002499 , 0.00012242, 0.00012253])
 'std_test_score': array([0.00413201, 0.00328157, 0.00303805, 0.00111611, 0.00110353])
 'std_train_score': array([6.81463842e-04, 5.30212406e-04, 2.92848128e-04, 8.33797837e
        0.00000000e+00])}
```

26.1 Cross Validation (CV): no need for separate validation set

By Fabian Flöck [CC BY-SA 3.0], from Wikimedia Commons

26.1.1 Varying regularization

```
In [127]: gridsearch = GridSearchCV(estimator=LogisticRegression(),
                            param_grid={'C': [0.1, 1, 10, 100], 'penalty': ['l1', 'l2']}, verbos

In [128]: gridsearch.fit(X_cvec, y)

Fitting 3 folds for each of 8 candidates, totalling 24 fits
[CV] C=0.1, penalty=l1 ...
[CV] ... C=0.1, penalty=l1, score=0.6701388888888888, total=   0.2s
[CV] C=0.1, penalty=l1 ...


[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.2s remaining:    0.0s


[CV] ... C=0.1, penalty=l1, score=0.6649642492339122, total=   0.2s
[CV] C=0.1, penalty=l1 ...


[Parallel(n_jobs=1)]: Done   2 out of   2 | elapsed:    0.4s remaining:    0.0s


[CV] ... C=0.1, penalty=l1, score=0.6740241160842019, total=   0.2s
[CV] C=0.1, penalty=l2 ...
[CV] ... C=0.1, penalty=l2, score=0.7757352941176471, total=   0.5s
[CV] C=0.1, penalty=l2 ...
[CV] ... C=0.1, penalty=l2, score=0.7662921348314606, total=   0.5s
[CV] C=0.1, penalty=l2 ...
[CV] ... C=0.1, penalty=l2, score=0.774167177600654, total=   0.4s
[CV] C=1, penalty=l1 ...
[CV] ... C=1, penalty=l1, score=0.7796160130718954, total=   0.2s
[CV] C=1, penalty=l1 ...
[CV] ... C=1, penalty=l1, score=0.7758937691521961, total=   0.2s
[CV] C=1, penalty=l1 ...
```

```
[CV] ... C=1, penalty=l1, score=0.7794808910688739, total=   0.3s
[CV] C=1, penalty=l2 ...
[CV] ... C=1, penalty=l2, score=0.8112745098039216, total=   1.0s
[CV] C=1, penalty=l2 ...
[CV] ... C=1, penalty=l2, score=0.8032686414708886, total=   0.8s
[CV] C=1, penalty=l2 ...
[CV] ... C=1, penalty=l2, score=0.8078888207643572, total=   0.8s
[CV] C=10, penalty=l1 ...
[CV] ... C=10, penalty=l1, score=0.784109477124183, total=   0.4s
[CV] C=10, penalty=l1 ...
[CV] ... C=10, penalty=l1, score=0.7697650663942799, total=   0.5s
[CV] C=10, penalty=l1 ...
[CV] ... C=10, penalty=l1, score=0.7780502759043532, total=   0.5s
[CV] C=10, penalty=l2 ...
[CV] ... C=10, penalty=l2, score=0.7998366013071896, total=   1.2s
[CV] C=10, penalty=l2 ...
[CV] ... C=10, penalty=l2, score=0.79244126659857, total=   1.3s
[CV] C=10, penalty=l2 ...
[CV] ... C=10, penalty=l2, score=0.7968526466380543, total=   1.2s
[CV] C=100, penalty=l1 ...
[CV] ... C=100, penalty=l1, score=0.7734885620915033, total=   0.5s
[CV] C=100, penalty=l1 ...
[CV] ... C=100, penalty=l1, score=0.760367722165475, total=   0.5s
[CV] C=100, penalty=l1 ...
[CV] ... C=100, penalty=l1, score=0.7600653995503781, total=   0.5s
[CV] C=100, penalty=l2 ...
[CV] ... C=100, penalty=l2, score=0.7792075163398693, total=   1.7s
[CV] C=100, penalty=l2 ...
[CV] ... C=100, penalty=l2, score=0.780388151174668, total=   1.6s
[CV] C=100, penalty=l2 ...
[CV] ... C=100, penalty=l2, score=0.7819333742080523, total=   1.4s


[Parallel(n_jobs=1)]: Done  24 out of  24 | elapsed:   16.9s finished


Out[128]: GridSearchCV(cv=None, error_score='raise',
                  estimator=LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercep
                     intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                     penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
                     verbose=0, warm_start=False),
                  fit_params=None, iid=True, n_jobs=1,
                  param_grid={'C': [0.1, 1, 10, 100], 'penalty': ['l1', 'l2']},
                  pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
                  scoring=None, verbose=3)

In [129]: gridsearch.best_params_

Out[129]: {'C': 1, 'penalty': 'l2'}
```

```
In [130]: gridsearch.best_score_

Out[130]: 0.8074775265595205

In [131]: gridsearch.cv_results_

Out[131]: {'mean_fit_time': array([0.2177523 , 0.46297407, 0.21892142, 0.84805981, 0.47023813,
                1.26045656, 0.50895095, 1.59414744]),
           'mean_score_time': array([0.00168649, 0.00156752, 0.00170739, 0.00163372, 0.00150959,
                0.00159375, 0.00137703, 0.00156943]),
           'mean_test_score': array([0.66970853, 0.77206483, 0.77833016, 0.80747753, 0.77730864,
                0.79637701, 0.76464179, 0.7805094 ]),
           'mean_train_score': array([0.69415719, 0.87806463, 0.92570155, 0.97950149, 0.99894445
                0.99819533, 1.        , 0.99979571]),
           'param_C': masked_array(data=[0.1, 0.1, 1, 1, 10, 10, 100, 100],
                        mask=[False, False, False, False, False, False, False, False],
                fill_value='?',
                     dtype=object),
           'param_penalty': masked_array(data=['l1', 'l2', 'l1', 'l2', 'l1', 'l2', 'l1', 'l2'],
                        mask=[False, False, False, False, False, False, False, False],
                fill_value='?',
                     dtype=object),
           'params': [{'C': 0.1, 'penalty': 'l1'},
            {'C': 0.1, 'penalty': 'l2'},
            {'C': 1, 'penalty': 'l1'},
            {'C': 1, 'penalty': 'l2'},
            {'C': 10, 'penalty': 'l1'},
            {'C': 10, 'penalty': 'l2'},
            {'C': 100, 'penalty': 'l1'},
            {'C': 100, 'penalty': 'l2'}],
           'rank_test_score': array([8, 6, 4, 1, 5, 2, 7, 3], dtype=int32),
           'split0_test_score': array([0.67013889, 0.77573529, 0.77961601, 0.81127451, 0.7841094
                0.7998366 , 0.77348856, 0.77920752]),
           'split0_train_score': array([0.69646506, 0.87832039, 0.92531671, 0.97987331, 0.999182
                0.99856968, 1.        , 0.99989783]),
           'split1_test_score': array([0.66496425, 0.76629213, 0.77589377, 0.80326864, 0.7697650
                0.79244127, 0.76036772, 0.78038815]),
           'split1_train_score': array([0.69486158, 0.87874144, 0.92777608, 0.97875166, 0.998876
                0.99785473, 1.        , 0.99979569]),
           'split2_test_score': array([0.67402412, 0.77416718, 0.77948089, 0.80788882, 0.7780502
                0.79685265, 0.7600654 , 0.78193337]),
           'split2_train_score': array([0.69114493, 0.87713206, 0.92401185, 0.97987948, 0.998774
                0.99816158, 1.        , 0.9996936 ]),
           'std_fit_time': array([0.0114245 , 0.01440613, 0.03147274, 0.09229467, 0.02759413,
                0.0570146 , 0.0238175 , 0.12339886]),
           'std_score_time': array([2.28217134e-04, 2.97928371e-04, 3.03991906e-04, 7.47829897e-
                1.65917558e-05, 8.08154192e-05, 5.99682631e-05, 7.09749147e-05]),
           'std_test_score': array([0.00371093, 0.00413201, 0.00172376, 0.00328157, 0.00588   ,
```

```
       0.00303805, 0.00625811, 0.00111611]),
'std_train_score': array([2.22831334e-03, 6.81463842e-04, 1.56064595e-03, 5.30212406e
       1.73510891e-04, 2.92848128e-04, 0.00000000e+00, 8.33797837e-05])}
```

- How to optimize hyperparameters of pipelines?
- This works automatically with pipelines of scikit-learn objects

27 Gridsearch on pipelines of scikit-learn objects

```
In [132]: pipeline = Pipeline(steps=  [   ('CountVectorizer', CountVectorizer()),
                                          ('NaiveBayes', MultinomialNB())          ])

In [133]: pipeline.get_params()

Out[133]: {'CountVectorizer': CountVectorizer(analyzer='word', binary=False, decode_error='stric
                  dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
                  lowercase=True, max_df=1.0, max_features=None, min_df=1,
                  ngram_range=(1, 1), preprocessor=None, stop_words=None,
                  strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
                  tokenizer=None, vocabulary=None),
           'CountVectorizer__analyzer': 'word',
           'CountVectorizer__binary': False,
           'CountVectorizer__decode_error': 'strict',
           'CountVectorizer__dtype': numpy.int64,
           'CountVectorizer__encoding': 'utf-8',
           'CountVectorizer__input': 'content',
           'CountVectorizer__lowercase': True,
           'CountVectorizer__max_df': 1.0,
           'CountVectorizer__max_features': None,
           'CountVectorizer__min_df': 1,
           'CountVectorizer__ngram_range': (1, 1),
           'CountVectorizer__preprocessor': None,
           'CountVectorizer__stop_words': None,
           'CountVectorizer__strip_accents': None,
           'CountVectorizer__token_pattern': '(?u)\\b\\w\\w+\\b',
           'CountVectorizer__tokenizer': None,
           'CountVectorizer__vocabulary': None,
           'NaiveBayes': MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True),
           'NaiveBayes__alpha': 1.0,
           'NaiveBayes__class_prior': None,
           'NaiveBayes__fit_prior': True,
           'memory': None,
           'steps': [('CountVectorizer',
             CountVectorizer(analyzer='word', binary=False, decode_error='strict',
                     dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
                     lowercase=True, max_df=1.0, max_features=None, min_df=1,
                     ngram_range=(1, 1), preprocessor=None, stop_words=None,
```

```
                    strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
                    tokenizer=None, vocabulary=None)),
            ('NaiveBayes', MultinomialNB(alpha=1.0, class_prior=None, fit_prior=True))]}
```

```
In [134]: MultinomialNB?
```

```
In [135]: param_grid = {  'CountVectorizer__binary': [True, False],
                          'CountVectorizer__ngram_range': [(1, 1), (1,2)],
                          'NaiveBayes__alpha': np.logspace(start=-1, stop=1, num=3)    }
```

```
In [136]: gridsearch = GridSearchCV(estimator=pipeline, param_grid=param_grid, verbose=5)
```

```
In [137]: gridsearch.fit(X, y)
```

```
Fitting 3 folds for each of 12 candidates, totalling 36 fits
[CV] CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=0.1
[CV]  CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=0.1,
[CV] CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=0.1


[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    0.7s remaining:    0.0s


[CV]  CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=0.1,
[CV] CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=0.1


[Parallel(n_jobs=1)]: Done   2 out of   2 | elapsed:    1.4s remaining:    0.0s


[CV]  CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=0.1,
[CV] CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=1.0


[Parallel(n_jobs=1)]: Done   3 out of   3 | elapsed:    2.1s remaining:    0.0s


[CV]  CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=1.0,
[CV] CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=1.0


[Parallel(n_jobs=1)]: Done   4 out of   4 | elapsed:    2.8s remaining:    0.0s


[CV]  CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=1.0,
[CV] CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=1.0
[CV]  CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=1.0,
[CV] CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=10.0
[CV]  CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=10.0,
[CV] CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=10.0
```

```
[CV]  CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=10.0,
[CV] CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=10.0
[CV]  CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=10.0,
[CV] CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=0.1
[CV]  CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=0.1,
[CV] CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=0.1
[CV]  CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=0.1,
[CV] CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=0.1
[CV]  CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=0.1,
[CV] CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=1.0
[CV]  CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=1.0,
[CV] CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=1.0
[CV]  CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=1.0,
[CV] CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=1.0
[CV]  CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=1.0,
[CV] CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=10.0
[CV]  CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=10.0,
[CV] CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=10.0
[CV]  CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=10.0,
[CV] CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=10.0
[CV]  CountVectorizer__binary=True, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=10.0,
[CV] CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=0.1
[CV]  CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=0.1,
[CV] CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=0.1
[CV]  CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=0.1,
[CV] CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=0.1
[CV]  CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=0.1,
[CV] CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=1.0
[CV]  CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=1.0,
[CV] CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=1.0
[CV]  CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=1.0,
[CV] CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=1.0
[CV]  CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=1.0,
[CV] CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=10.0
[CV]  CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=10.0
[CV] CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=10.0
[CV]  CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=10.0
[CV] CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=10.0
[CV]  CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 1), NaiveBayes__alpha=10.0
[CV] CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=0.1
[CV]  CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=0.1,
[CV] CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=0.1
[CV]  CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=0.1,
[CV] CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=0.1
[CV]  CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=0.1,
[CV] CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=1.0
[CV]  CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=1.0,
[CV] CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=1.0
```

```
[CV]  CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=1.0,
[CV] CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=1.0
[CV]  CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=1.0,
[CV] CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=10.0
[CV]  CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=10.0
[CV] CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=10.0
[CV]  CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=10.0
[CV] CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=10.0
[CV]  CountVectorizer__binary=False, CountVectorizer__ngram_range=(1, 2), NaiveBayes__alpha=10.0


[Parallel(n_jobs=1)]: Done  36 out of  36 | elapsed:   44.6s finished
```

```
Out[137]: GridSearchCV(cv=None, error_score='raise',
                 estimator=Pipeline(memory=None,
               steps=[('CountVectorizer', CountVectorizer(analyzer='word', binary=False, decode_
                  dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
                  lowercase=True, max_df=1.0, max_features=None, min_df=1,
                  ngram_range=(1, 1), p...one, vocabulary=None)), ('NaiveBayes', MultinomialNB(a
                 fit_params=None, iid=True, n_jobs=1,
                 param_grid={'CountVectorizer__binary': [True, False], 'CountVectorizer__ngram_r
                 pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
                 scoring=None, verbose=5)
```

```
In [138]: gridsearch.best_score_
```

```
Out[138]: 0.8397575592481613
```

```
In [139]: gridsearch.best_params_
```

```
Out[139]: {'CountVectorizer__binary': True,
           'CountVectorizer__ngram_range': (1, 2),
           'NaiveBayes__alpha': 0.1}
```

```
In [140]: gridsearch.cv_results_
```

```
Out[140]: {'mean_fit_time': array([0.30203183, 0.29733133, 0.26862709, 1.04726974, 1.09419902,
                  1.02227044, 0.26516604, 0.25951457, 0.28823471, 0.9809711 ,
                  0.98117526, 1.02250608]),
           'mean_score_time': array([0.13935637, 0.1388905 , 0.11800853, 0.24567167, 0.25666237,
                  0.25152214, 0.11827564, 0.12116877, 0.13475227, 0.25594536,
                  0.2561365 , 0.26719125]),
           'mean_test_score': array([0.83417325, 0.83090439, 0.7246663 , 0.83975756, 0.82913375,
                  0.70103514, 0.83512667, 0.82899755, 0.70757287, 0.83887224,
                  0.82504767, 0.66984473]),
           'mean_train_score': array([0.95215891, 0.92764245, 0.79807968, 0.99785482, 0.9899891
                  0.86764519, 0.94834534, 0.92178569, 0.77397189, 0.99775269,
                  0.98842278, 0.83369682]),
```

```
'param_CountVectorizer__binary': masked_array(data=[True, True, True, True, True, Tru
                   False, False, False, False],
             mask=[False, False, False, False, False, False, False, False,
                   False, False, False, False],
       fill_value='?',
            dtype=object),
'param_CountVectorizer__ngram_range': masked_array(data=[(1, 1), (1, 1), (1, 1), (1,
                   (1, 1), (1, 1), (1, 2), (1, 2), (1, 2)],
             mask=[False, False, False, False, False, False, False, False,
                   False, False, False, False],
       fill_value='?',
            dtype=object),
'param_NaiveBayes__alpha': masked_array(data=[0.1, 1.0, 10.0, 0.1, 1.0, 10.0, 0.1, 1.
                   1.0, 10.0],
             mask=[False, False, False, False, False, False, False, False,
                   False, False, False, False],
       fill_value='?',
            dtype=object),
'params': [{'CountVectorizer__binary': True,
  'CountVectorizer__ngram_range': (1, 1),
  'NaiveBayes__alpha': 0.1},
 {'CountVectorizer__binary': True,
  'CountVectorizer__ngram_range': (1, 1),
  'NaiveBayes__alpha': 1.0},
 {'CountVectorizer__binary': True,
  'CountVectorizer__ngram_range': (1, 1),
  'NaiveBayes__alpha': 10.0},
 {'CountVectorizer__binary': True,
  'CountVectorizer__ngram_range': (1, 2),
  'NaiveBayes__alpha': 0.1},
 {'CountVectorizer__binary': True,
  'CountVectorizer__ngram_range': (1, 2),
  'NaiveBayes__alpha': 1.0},
 {'CountVectorizer__binary': True,
  'CountVectorizer__ngram_range': (1, 2),
  'NaiveBayes__alpha': 10.0},
 {'CountVectorizer__binary': False,
  'CountVectorizer__ngram_range': (1, 1),
  'NaiveBayes__alpha': 0.1},
 {'CountVectorizer__binary': False,
  'CountVectorizer__ngram_range': (1, 1),
  'NaiveBayes__alpha': 1.0},
 {'CountVectorizer__binary': False,
  'CountVectorizer__ngram_range': (1, 1),
  'NaiveBayes__alpha': 10.0},
 {'CountVectorizer__binary': False,
  'CountVectorizer__ngram_range': (1, 2),
  'NaiveBayes__alpha': 0.1},
```

```
 {'CountVectorizer__binary': False,
  'CountVectorizer__ngram_range': (1, 2),
  'NaiveBayes__alpha': 1.0},
 {'CountVectorizer__binary': False,
  'CountVectorizer__ngram_range': (1, 2),
  'NaiveBayes__alpha': 10.0}],
'rank_test_score': array([ 4,  5,  9,  1,  6, 11,  3,  7, 10,  2,  8, 12], dtype=int3
'split0_test_score': array([0.83517157, 0.83394608, 0.73549837, 0.84232026, 0.8331290
       0.71058007, 0.83721405, 0.83272059, 0.7191585 , 0.84313725,
       0.82986111, 0.67851307]),
'split0_train_score': array([0.95422967, 0.92991418, 0.80138946, 0.99785452, 0.990396
       0.86881896, 0.95167552, 0.92439722, 0.7772783 , 0.99785452,
       0.98876175, 0.83673886]),
'split1_test_score': array([0.83125638, 0.82410623, 0.72604699, 0.83309499, 0.8286006
       0.70234934, 0.8308478 , 0.82574055, 0.70745659, 0.83309499,
       0.82226762, 0.66945863]),
'split1_train_score': array([0.95055675, 0.92603943, 0.79517826, 0.99805905, 0.989273
       0.86791296, 0.94626622, 0.9189907 , 0.77229543, 0.99805905,
       0.98804781, 0.83328226]),
'split2_test_score': array([0.83609238, 0.83466176, 0.71244635, 0.84385857, 0.8256693
       0.69016963, 0.83731862, 0.82853055, 0.69609646, 0.84038422,
       0.82301247, 0.66155733]),
'split2_train_score': array([0.95169033, 0.92697375, 0.79767133, 0.9976509 , 0.990297
       0.86620366, 0.94709427, 0.92196916, 0.77234195, 0.9973445 ,
       0.98845879, 0.83106935]),
'std_fit_time': array([0.00854737, 0.0143666 , 0.00437476, 0.05998347, 0.06358373,
       0.03809166, 0.00705312, 0.00488276, 0.01307075, 0.04304892,
       0.02408205, 0.06969686]),
'std_score_time': array([0.0052155 , 0.00357564, 0.00632402, 0.00991197, 0.01498425,
       0.01137845, 0.0060038 , 0.00531616, 0.00397409, 0.0300545 ,
       0.01537111, 0.03533054]),
'std_test_score': array([0.00209662, 0.00481614, 0.0094613 , 0.00475306, 0.00306863,
       0.00838405, 0.00302607, 0.00286889, 0.00941523, 0.00423712,
       0.00341786, 0.00692742]),
'std_train_score': array([0.00153564, 0.00165102, 0.0025521 , 0.00016662, 0.0005075 ,
       0.00108435, 0.00237894, 0.00221101, 0.00233806, 0.00030047,
       0.00029257, 0.00233306])}
```

In [141]:
```
pipeline.set_params(**gridsearch.best_params_);
```

In [142]:
```
pipeline.fit(X, y);
```

In [143]:
```
pipeline.score(X, y)
```

Out[143]:
```
0.9969354399346227
```

In [144]:
```
pipeline.score(X_val, y_val)
```

Out[144]:
```
0.8473953013278857
```

28 Adding non-sklearn objects

Gridsearch ⇒ derive from `BaseEstimator`
`fit_transform` ⇒ derive from `TransformerMixin`

```
In [145]: from sklearn.base import BaseEstimator
          from sklearn.base import TransformerMixin

In [146]: class GensimWord2Vec(TransformerMixin, BaseEstimator): # Derive from BaseEstimator!
              def __init__(self, size=100, min_count=5):
                  self.size=size
                  self.min_count=min_count

              def fit(self, X, y=None):
                  self.model = Word2Vec(X, size=self.size, min_count=self.min_count)
                  return self


              def transform(self, X, y=None):
                  lower = X.str.lower()
                  tokenized = lower.str.split(' ')
                  vectors = tokenized.apply(lambda l: [self.model[word] for word in l if word in
                  def average(l):
                      if l == []:
                          return np.zeros(self.model.vector_size)
                      else:
                          return np.mean(l, axis=0)
                  vectors = vectors.apply(average)
                  vectors = vectors.apply(pd.Series)
                  return vectors

In [147]: pipeline = Pipeline(steps=  [  ('word2vec', GensimWord2Vec()),
                                         ('logreg', LogisticRegression())        ])

In [148]: pipeline.get_params()

Out[148]: {'logreg': LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True
                     intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                     penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
                     verbose=0, warm_start=False),
           'logreg__C': 1.0,
           'logreg__class_weight': None,
           'logreg__dual': False,
           'logreg__fit_intercept': True,
           'logreg__intercept_scaling': 1,
           'logreg__max_iter': 100,
           'logreg__multi_class': 'ovr',
           'logreg__n_jobs': 1,
           'logreg__penalty': 'l2',
```

```
            'logreg__random_state': None,
            'logreg__solver': 'liblinear',
            'logreg__tol': 0.0001,
            'logreg__verbose': 0,
            'logreg__warm_start': False,
            'memory': None,
            'steps': [('word2vec', GensimWord2Vec(min_count=5, size=100)),
             ('logreg',
              LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
                        intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                        penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
                        verbose=0, warm_start=False))],
            'word2vec': GensimWord2Vec(min_count=5, size=100),
            'word2vec__min_count': 5,
            'word2vec__size': 100}

In [149]: param_grid = {   'word2vec__min_count': [1],
                           'word2vec__size': [10, 50]           }

In [150]: gridsearch = GridSearchCV(estimator=pipeline, param_grid=param_grid, verbose=5)

In [151]: gridsearch.fit(X, y)

Fitting 3 folds for each of 2 candidates, totalling 6 fits
[CV] word2vec__min_count=1, word2vec__size=10 ...
[CV]  word2vec__min_count=1, word2vec__size=10, score=0.40951797385620914, total=   5.0s
[CV] word2vec__min_count=1, word2vec__size=10 ...


[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:    7.1s remaining:    0.0s


[CV]  word2vec__min_count=1, word2vec__size=10, score=0.4212461695607763, total=   4.5s
[CV] word2vec__min_count=1, word2vec__size=10 ...


[Parallel(n_jobs=1)]: Done   2 out of   2 | elapsed:   13.7s remaining:    0.0s


[CV]  word2vec__min_count=1, word2vec__size=10, score=0.41630901287553645, total=   4.5s
[CV] word2vec__min_count=1, word2vec__size=50 ...


[Parallel(n_jobs=1)]: Done   3 out of   3 | elapsed:   20.2s remaining:    0.0s


[CV]  word2vec__min_count=1, word2vec__size=50, score=0.4103349673202614, total=   4.8s
[CV] word2vec__min_count=1, word2vec__size=50 ...
```

```
[Parallel(n_jobs=1)]: Done   4 out of   4 | elapsed:   27.1s remaining:    0.0s


[CV]  word2vec__min_count=1, word2vec__size=50, score=0.4218590398365679, total=   4.8s
[CV] word2vec__min_count=1, word2vec__size=50 ...
[CV]  word2vec__min_count=1, word2vec__size=50, score=0.41630901287553645, total=   4.8s


[Parallel(n_jobs=1)]: Done   6 out of   6 | elapsed:   40.9s finished


Out[151]: GridSearchCV(cv=None, error_score='raise',
                  estimator=Pipeline(memory=None,
                 steps=[('word2vec', GensimWord2Vec(min_count=5, size=100)), ('logreg', LogisticRe
                     intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
                     penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
                     verbose=0, warm_start=False))]),
                  fit_params=None, iid=True, n_jobs=1,
                  param_grid={'word2vec__min_count': [1], 'word2vec__size': [10, 50]},
                  pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
                  scoring=None, verbose=5)

In [152]: gridsearch.best_params_

Out[152]: {'word2vec__min_count': 1, 'word2vec__size': 50}

In [153]: gridsearch.best_score_

Out[153]: 0.4161672568782348

In [154]: pipeline.set_params(**gridsearch.best_params_);

In [155]: pipeline.fit(X, y);

In [156]: pipeline.score(X, y)

Out[156]: 0.41664396622173794

In [157]: pipeline.score(X_val, y_val)

Out[157]: 0.4071501532175689
```

29 Conclusion

- **Pipelines** ⇒ **clear code** + **easy experiments**
- **Gridsearch** ⇒ **tuning** of **hyperparameters**
- **Scikit-learn** ⇒ convenient classes for both

30 Thank you for your attention