# A Practical Introduction to Machine Learning in Python
# Day 4 – Thursday
# »Supervised Machine Learning«

Rupert Kiddle
Marieke van Hoof

r.t.kiddle@vu.nl, @rptkiddle
m.vanhoof@uva.nl, @marieke_vh

September 19, 2024

# Today

The problem of overfitting

- Train/validation/test split
- Cross-validation

Finding the optimal (hyper-)parameters

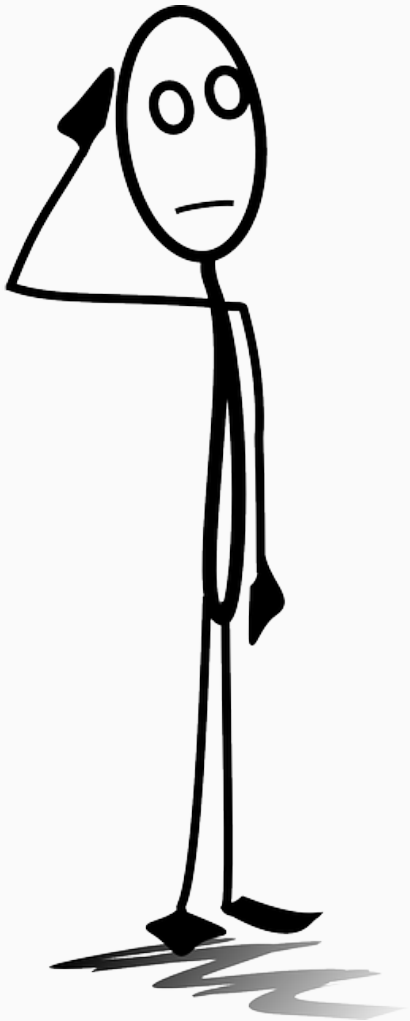
- Grid search
- A typical case for gridsearch: The regularization parameter C

More suggestions for improving your models

- Tuning decision thresholds with ROC curves

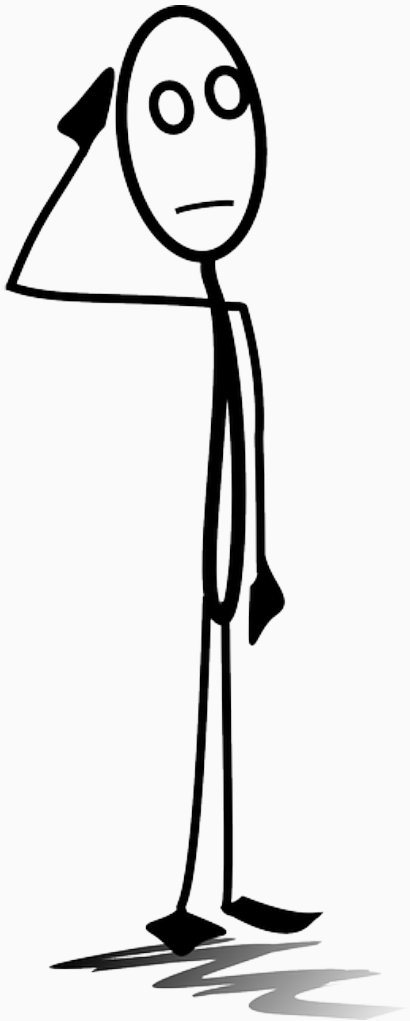
Exercise

# Also read chapter 8.5!



*Everything clear from this morning?*

# The problem of overfitting



*Isn't training multiple models and then selecting the best one a bit like p-hacking?*

# Yes. We call this problem “overfitting”.

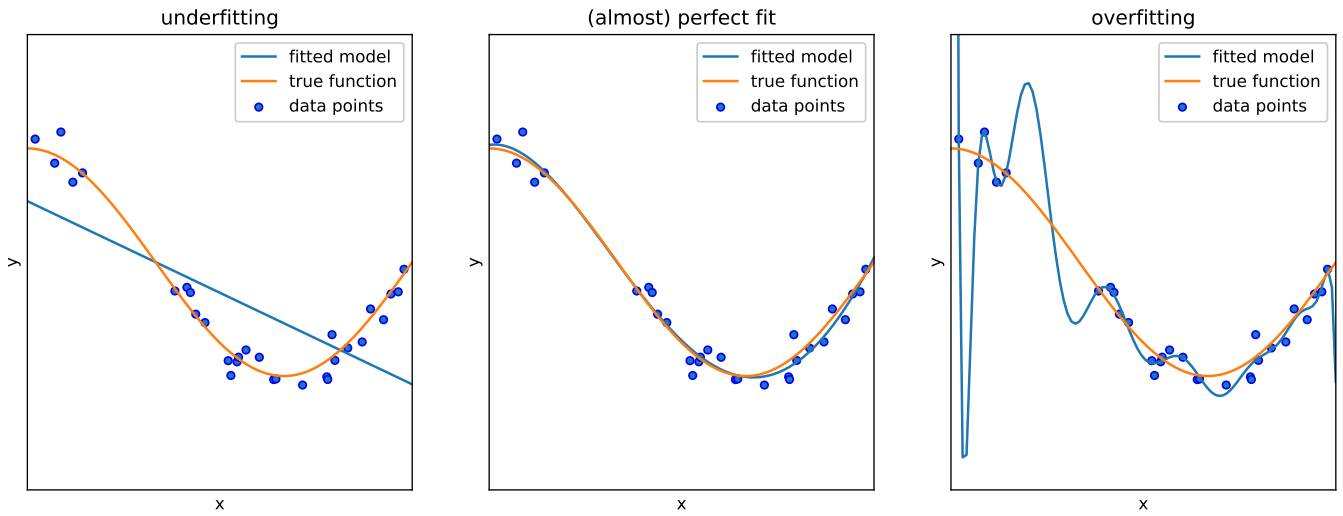



**Figure 1:** Underfitting and overfitting. Example adapted from https://scikit-learn.org/stable/auto_examples/model_selection/plot_underfitting_overfitting

# How to avoid overfitting

1. **Train/test split.** We already do this. It avoids overfitting on the training data.[1]
2. **Train/validation/test split.** But maybe we overfit on the test data now? We could set aside *third* dataset.
3. ***k*-fold crossvalidation.** Extending the above such that every case is sometimes $k-1$ times part of the training data and 1 time part of the validation data.
4. **Regularization.** (in combination with the above) We "penalize" models for being too complex.

[1] In classical statistics, the $R^2$ of an OLS regression is prone to that. Calculating $R^2$ on a separate test set would be much more conservative.

# How to avoid overfitting

1. **Train/test split.** We already do this. It avoids overfitting on the training data.[1]
2. **Train/validation/test split.** But maybe we overfit on the test data now? We could set aside *third* dataset.
3. **$k$-fold crossvalidation.** Extending the above such that every case is sometimes $k-1$ times part of the training data and 1 time part of the validation data.
4. **Regularization.** (in combination with the above) We "penalize" models for being too complex.

[1] In classical statistics, the $R^2$ of an OLS regression is prone to that. Calculating $R^2$ on a separate test set would be much more conservative.

# How to avoid overfitting

1. **Train/test split.** We already do this. It avoids overfitting on the training data.[1]
2. **Train/validation/test split.** But maybe we overfit on the test data now? We could set aside *third* dataset.
3. ***k*-fold crossvalidation.** Extending the above such that every case is sometimes $k-1$ times part of the training data and 1 time part of the validation data.
4. **Regularization.** (in combination with the above) We "penalize" models for being too complex.

[1] In classical statistics, the $R^2$ of an OLS regression is prone to that. Calculating $R^2$ on a separate test set would be much more conservative.

# How to avoid overfitting

1. **Train/test split.** We already do this. It avoids overfitting on the training data.[1]
2. **Train/validation/test split.** But maybe we overfit on the test data now? We could set aside *third* dataset.
3. ***k*-fold crossvalidation.** Extending the above such that every case is sometimes $k-1$ times part of the training data and 1 time part of the validation data.
4. **Regularization.** (in combination with the above) We "penalize" models for being too complex.

[1] In classical statistics, the $R^2$ of an OLS regression is prone to that. Calculating $R^2$ on a separate test set would be much more conservative.

# The problem of overfitting

## Train/validation/test split

# Train/validation/test split

- When you compare a lot of different models (or (hyper-)parameters), you might want to evaluate (compare) them using a third dataset
- e.g., make 80/20 split (train/test); then split first part again 80/20 (train/validation)
- only use the test data *at the very end* to get a final estimate of how good your model is.

In short: Validation data to *select* the best approach; test data to get the accuracy of the approach you chose.

# Train/validation/test split

- When you compare a lot of different models (or (hyper-)parameters), you might want to evaluate (compare) them using a third dataset
- e.g., make 80/20 split (train/test); then split first part again 80/20 (train/validation)
- only use the test data *at the very end* to get a final estimate of how good your model is.

In short: Validation data to *select* the best approach; test data to get the accuracy of the approach you chose.

# The problem of overfitting

## Cross-validation

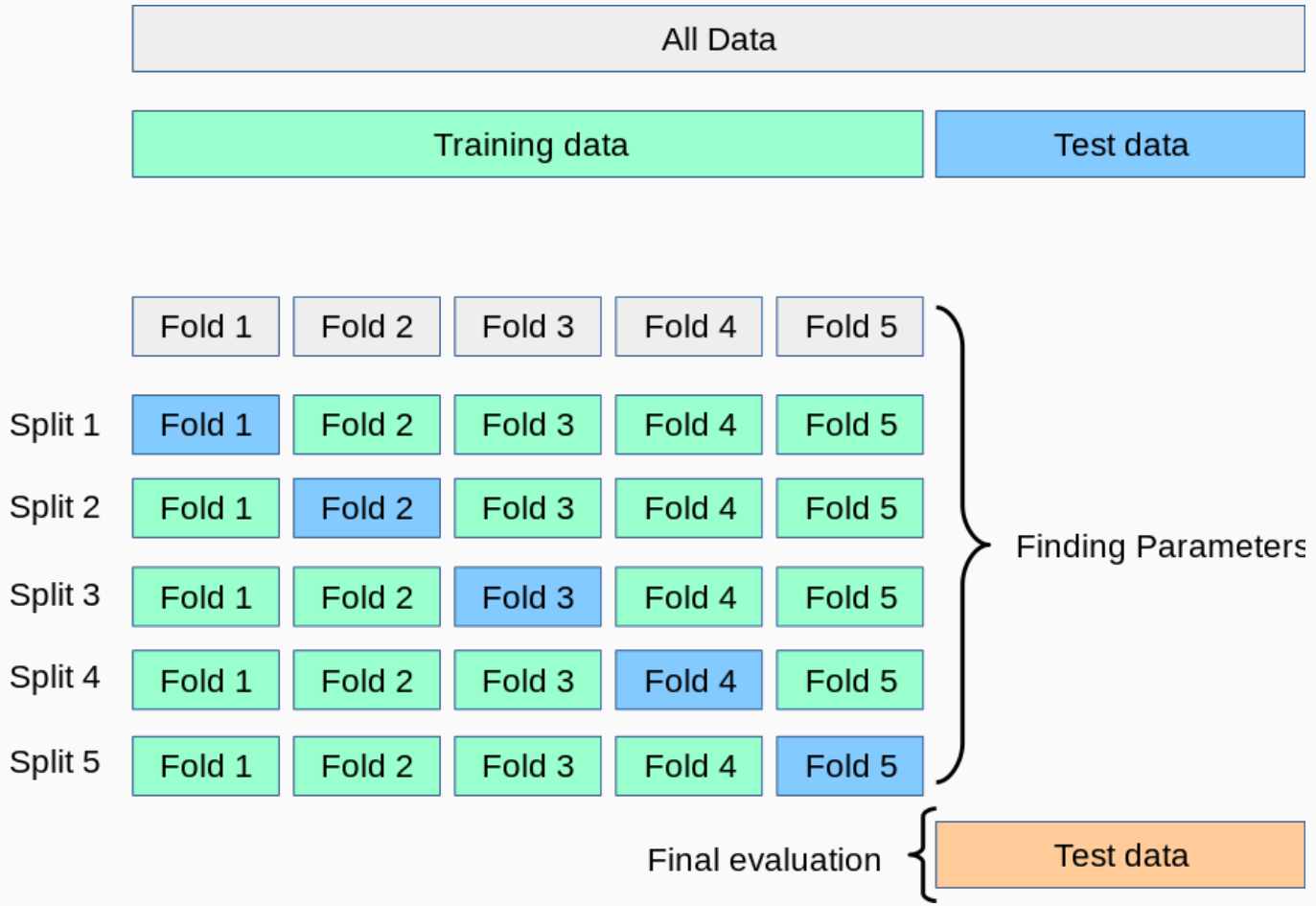



**Figure 2:** First, we set aside a test dataset for the final evaluation. Then, we split our data into $k = 5$ folds, where each fold is used for validation once (blue) and $k - 1 = 4$ times for training. Source: https://scikit-learn.org/stable/_images/grid_search_cross_validation.png

# Cross-validation

```
1 from sklearn.model_selection import cross_val_score
2 from sklearn.naive_bayes import MultinomialNB
3 nb = MultinomialNB()  # the classifier we trained this week
4 scores = cross_val_score(nb, train_features, [r[1] for r in reviews], cv
      =10)
5 print(scores)
```

results in:

```
1 [0.858 0.8612 0.8516 0.8528 0.8672 0.8664 0.8576 0.8652 0.8436 0.852 ]
```

We estimate the model 10 times on different training/validation data splits and get 10 different evaluation scores (here: accuracy, but we can use precision, recall, F1, etc. – see examples in the book).

Note that a simple 50:50 train/validation split is identical to setting $k = 2$

# Reasons to do Cross-validation

## We can get confidence intervals around the scores

- If we have 10 scores instead of one, we can not only get the mean, but also a standard deviation
- If you have two models, both with a mean accuracy (or F1, or whatever) of .85 but one with a large and one with a low standard deviation, you probably prefer the latter – it generalizes better, less likely to suffer from overfitting

# Reasons to do Cross-validation

**We do not "waste" too much validation data**

- If $k = 10$ (the most typical value), in each iteration, we use 90% of the data for training.
- We even use *all* data (except test set, of course) for training at least once.
- With train/validation split, we probably need a larger validation set to be sure (e.g., 80/20)
- Especially relevant when annotation is expensive.

# Takeaway i

**Simple train/test split**

- introductory examples/pedagical reasons
- really small dataset where you cannot afford to set aside validation data
- if – for whatever reason – you do not compare different models and settings.

# Takeaway ii

**Train/validation/test split**

- To compare different model configurations without overfitting on the test data
- Pedacogical reasons, simple start
- Very large datasets or very complex models where setting $k > 2$ would lead to prohibtitively expensive computations

# Takeaway iii

**$k$-fold crossvalidation**

- Best option for comparing multiple models, *especially* when engaging in *hyperparameter tuning* (next section)

# Takeaway iv

**Also a nice example for changing standards in the field**

A couple of years ago, you could get away with just doing a train/test split in your paper. Nowadays, chances are high a reviewer will reject this.

In other words: whether you have or create a train/validation/test or use cross-validation – just make sure you always set aside one test set for final evaluation that you have never used before.

# Finding the optimal (hyper-)parameters

**hyperparameter** a parameter of a model that is not learned through training, but specified in advance

# Finding the optimal (hyper-)parameters

## Grid search

# Hyperparameter optimization with grid search

**General idea**

Rather than arbitrary trying some configurations, let's systematically test and compare.

First idea: Use a for-loop! (Example 11.4 in the book)

```
configurations =
[('NB with Count', CountVectorizer(min_df=5, max_df=.5), MultinomialNB()
    ),
('NB with TfIdf', TfidfVectorizer(min_df=5, max_df=.5), MultinomialNB())
    ,
('LogReg with Count', CountVectorizer(min_df=5, max_df=.5),
    LogisticRegression(solver='liblinear')),
('LogReg with TfIdf', TfidfVectorizer(min_df=5, max_df=.5),
    LogisticRegression(solver='liblinear'))]

for description, vectorizer, classifier in configurations:
    print(description)
    X_train = vectorizer.fit_transform(reviews_train)
    X_test = vectorizer.transform(reviews_test)
    classifier.fit(X_train, y_train)
    y_pred = classifier.predict(X_test)
    short_classification_report(y_test, y_pred)
    print('\n')
```

(where (X_test, y_test) hopefully refers to a validation dataset with another test dataset set aside)

# A wishlist

We now tested $2 \times 2$ combinations: (NB|LogReg) $\times$ (count|tf· idf).

Wouldn't it be nice if we...

- could simply state this instead of manually creating the list?
- (especially if we had (NB|LogReg) $\times$ (count|tf· idf) $\times$ (min_df=5|min_df=1) $\times$ (max_df=.5|max_df=.8 |max_df=.9) $= 2 \times 2 \times 2 \times 3 = 24$ options)
- would not have to check manually which one performed best?

**That's what scikit-learn's gridsearch functionality does!** (and the second bullet point is called the *grid*)

# A wishlist

We now tested $2 \times 2$ combinations: (NB|LogReg) $\times$ (count|tf· idf).

Wouldn't it be nice if we...

- could simply state this instead of manually creating the list?
- (especially if we had (NB|LogReg) $\times$ (count|tf· idf) $\times$ (min_df=5|min_df=1) $\times$ (max_df=.5|max_df=.8|max_df=.9) $= 2 \times 2 \times 2 \times 3 = 24$ options)
- would not have to check manually which one performed best?

**That's what scikit-learn's gridsearch functionality does!** (and the second bullet point is called the *grid*)

# A wishlist

We now tested $2 \times 2$ combinations: (NB|LogReg) $\times$ (count|tf$\cdot$ idf).

Wouldn't it be nice if we...

- could simply state this instead of manually creating the list?
- (especially if we had (NB|LogReg) $\times$ (count|tf$\cdot$ idf) $\times$ (min_df=5|min_df=1) $\times$ (max_df=.5|max_df=.8 |max_df=.9) $= 2 \times 2 \times 2 \times 3 = 24$ options)
- would not have to check manually which one performed best?

**That's what scikit-learn's gridsearch functionality does!** (and the second bullet point is called the *grid*)

# A wishlist

We now tested $2 \times 2$ combinations: (NB|LogReg) $\times$ (count|tf$\cdot$ idf).

Wouldn't it be nice if we...

- could simply state this instead of manually creating the list?
- (especially if we had (NB|LogReg) $\times$ (count|tf$\cdot$ idf) $\times$ (min_df=5|min_df=1) $\times$ (max_df=.5|max_df=.8 |max_df=.9) $= 2 \times 2 \times 2 \times 3 = 24$ options)
- would not have to check manually which one performed best?

**That's what scikit-learn's gridsearch functionality does!** (and the second bullet point is called the *grid*)

# A wishlist

We now tested $2 \times 2$ combinations: (NB|LogReg) $\times$ (count|tf$\cdot$ idf).

Wouldn't it be nice if we...

- could simply state this instead of manually creating the list?
- (especially if we had (NB|LogReg) $\times$ (count|tf$\cdot$ idf) $\times$ (min_df=5|min_df=1) $\times$ (max_df=.5|max_df=.8 |max_df=.9) $= 2 \times 2 \times 2 \times 3 = 24$ options)
- would not have to check manually which one performed best?

**That's what scikit-learn's gridsearch functionality does!** (and the second bullet point is called the *grid*)

Gridsearch is especially useful for hyperparameter optimization, such as trying out different values for min_df and max_df on the previous slide.

You could also use a combined approach where you first use a handwritten loop to narrow down the number of candidate models and then tune the model(s) you settled on with a grid search.

# Finding the optimal (hyper-)parameters

## A typical case for gridsearch: The regularization parameter C

# The regularization parameter $\lambda$ (or $C = .5n\lambda$)

$$\arg\min_{\boldsymbol{\beta}} \left[ \sum_{i=1}^{n} \left( Y_i - \beta_0 - \sum_{j=1}^{p} \beta_j X_{ji} \right)^2 + \lambda \sum_{j=1}^{p} |\beta_j|^q \right]$$

We estimate the $\beta$-coefficients of a model by optimizing a so-called loss function, i.e minimizing the "cost" that occurs when the prediction is wrong.

*If we simply add the sum of the $\beta$ coefficients* to the model, we "punish" large coefficients.

If $\lambda$ is larger, we punish more.

If $q = 1$, we call it L1 regularization. If we add the squared coefficients ($q = 2$), we call it L2 regularization.

# The regularization parameter $\lambda$ (or $C = .5n\lambda$)

$$\arg\min_{\boldsymbol{\beta}} \left[ \sum_{i=1}^{n} \left( Y_i - \beta_0 - \sum_{j=1}^{p} \beta_j X_{ji} \right)^2 + \lambda \sum_{j=1}^{p} |\beta_j|^q \right]$$

We estimate the $\beta$-coefficients of a model by optimizing a so-called loss function, i.e minimizing the "cost" that occurs when the prediction is wrong.

*If we simply add the sum of the $\beta$ coefficients* to the model, we "punish" large coefficients.

If $\lambda$ is larger, we punish more.

If $q = 1$, we call it L1 regularization. If we add the squared coefficients ($q = 2$), we call it L2 regularization.

# The regularization parameter $\lambda$ (or $C = .5n\lambda$)

- scikit-learn uses regularization by default
- (that's why the coefficients of a logistic regression classifier will differ from a "traditional" logistic regression with the statsmodels module)
- Remember that we are *not* interested in a substantive interpretation of the $\beta$-coefficients but in a good prediction (which is why "punishing" large coefficients and hence biasing the coeffients is not a problem at all!)
- The penalty will shrink the coefficients towards zero, *which is very useful to avoid overfitting!*
- As always, there is a lot of extra info on the scikit-learn website

# The regularization parameter $\lambda$ (or $C = .5n\lambda$)

- scikit-learn uses regularization by default
- (that's why the coefficients of a logistic regression classifier will differ from a "traditional" logistic regression with the statsmodels module)
- Remember that we are *not* interested in a substantive interpretation of the $\beta$-coefficients but in a good prediction (which is why "punishing" large coefficients and hence biasing the coeffients is not a problem at all!)
- The penalty will shrink the coefficients towards zero, *which is very useful to avoid overfitting!*
- As always, there is a lot of extra info on the scikit-learn website

# The regularization parameter $\lambda$ (or $C = .5n\lambda$)

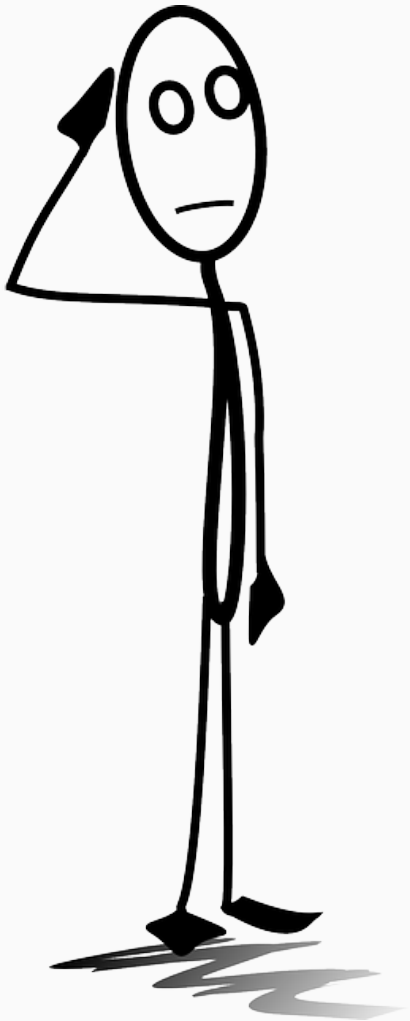
- scikit-learn uses regularization by default
- (that's why the coefficients of a logistic regression classifier will differ from a "traditional" logistic regression with the statsmodels module)
- Remember that we are *not* interested in a substantive interpretation of the $\beta$-coefficients but in a good prediction (which is why "punishing" large coefficients and hence biasing the coeffients is not a problem at all!)
- The penalty will shrink the coefficients towards zero, *which is very useful to avoid overfitting!*
- As always, there is a lot of extra info on the scikit-learn website

# The regularization parameter $\lambda$ (or $C = .5n\lambda$)

- scikit-learn uses regularization by default
- (that's why the coefficients of a logistic regression classifier will differ from a "traditional" logistic regression with the statsmodels module)
- Remember that we are *not* interested in a substantive interpretation of the $\beta$-coefficients but in a good prediction (which is why "punishing" large coefficients and hence biasing the coeffients is not a problem at all!)
- The penalty will shrink the coefficients towards zero, *which is very useful to avoid overfitting!*
- As always, there is a lot of extra info on the scikit-learn website

# The regularization parameter $\lambda$ (or $C = .5n\lambda$)

- scikit-learn uses regularization by default
- (that's why the coefficients of a logistic regression classifier will differ from a "traditional" logistic regression with the statsmodels module)
- Remember that we are *not* interested in a substantive interpretation of the $\beta$-coefficients but in a good prediction (which is why "punishing" large coefficients and hence biasing the coeffients is not a problem at all!)
- The penalty will shrink the coefficients towards zero, *which is very useful to avoid overfitting!*
- As always, there is a lot of extra info on the scikit-learn website



*How harsh should the penalty be?*

# Hyperparameter optimization with grid search

## General idea

Rather than arbitrary trying some combinations of hyperparameters, let's systematically test and compare.

## Example

# Hyperparameter optimization with grid search

## General idea

Rather than arbitrary trying some combinations of hyperparameters, let's systematically test and compare.

## Example

- To avoid overfitting, scikit-learn adds a *regularization term* to the loss function that is minimized to fit the regression.
- Think of this term as a penalty for overfitting
- How much weight should our penalty carry? That's determined by a constant, $C$.
- How to determine the best $C$? $\Rightarrow$ grid search

# Hyperparameter optimization with grid search

## General idea

Rather than arbitrary trying some combinations of hyperparameters, let's systematically test and compare.

## Example

- To avoid overfitting, scikit-learn adds a *regularization term* to the loss function that is minimized to fit the regression.
- Think of this term as a penalty for overfitting
- How much weight should our penalty carry? That's determined by a constant, $C$.
- How to determine the best $C$? $\Rightarrow$ grid search

# Hyperparameter optimization with grid search

## General idea

Rather than arbitrary trying some combinations of hyperparameters, let's systematically test and compare.

## Example

- To avoid overfitting, scikit-learn adds a *regularization term* to the loss function that is minimized to fit the regression.
- Think of this term as a penalty for overfitting
- How much weight should our penalty carry? That's determined by a constant, $C$.
- How to determine the best $C$? $\Rightarrow$ grid search

# Hyperparameter optimization with grid search

## General idea

Rather than arbitrary trying some combinations of hyperparameters, let's systematically test and compare.

## Example

- To avoid overfitting, scikit-learn adds a *regularization term* to the loss function that is minimized to fit the regression.
- Think of this term as a penalty for overfitting
- How much weight should our penalty carry? That's determined by a constant, $C$.
- How to determine the best $C$? $\Rightarrow$ grid search

# Hyperparameter optimization with grid search

Finding $C$ in a logistic regression using 5-fold cross-validation

```
1 from sklearn.linear_model import LogisticRegressionCV
2 logregCV = LogisticRegressionCV(cv=5).fit(train_features, [r[1] for r in
      reviews])
```

- Here, we just need to use `LogisticRegressionCV` instead of `LogisticRegression`
- But we can use it to test any combination of choices

# Hyperparameter optimization with grid search

Finding $C$ in a logistic regression using 5-fold cross-validation

```
1 from sklearn.linear_model import LogisticRegressionCV
2 logregCV = LogisticRegressionCV(cv=5).fit(train_features, [r[1] for r in
      reviews])
```

- Here, we just need to use `LogisticRegressionCV` instead of `LogisticRegression`.
- But we can use it to test any combination of choices

# Hyperparameter optimization with grid search

Finding $C$ in a logistic regression using 5-fold cross-validation

```
1 from sklearn.linear_model import LogisticRegressionCV
2 logregCV = LogisticRegressionCV(cv=5).fit(train_features, [r[1] for r in
      reviews])
```

- Here, we just need to use `LogisticRegressionCV` instead of `LogisticRegression`.
- But we can use it to test any combination of choices

```
pipeline = Pipeline(steps = [('vectorizer', TfidfVectorizer()), ('
    classifier', LogisticRegression(solver='liblinear'))])
grid = {
    'vectorizer__ngram_range' : [(1,1), (1,2)],
    'vectorizer__max_df': [0.5, 1.0],
    'vectorizer__min_df': [0, 5],
    'classifier__C': [0.01, 1, 100]
}

search = GridSearchCV(estimator=pipeline,
                      param_grid=grid,
                      scoring='accuracy',
                      cv=5,
                      n_jobs=-1, # use all cpus
                      verbose=10)
search.fit(reviews_train, y_train)
print(f'Using these hyperparameters {search.best_params_}, we get the
    best performance:')
print(short_classification_report(y_test, search.predict(reviews_test)))
```

Example 11.6 (book)

Note that the grid is specified as a dictionary where the keys are strings with the name of the step in the pipeline followed by two underscores followed by the parameter to tune, and the values are lists of values.

This means that *any* parameter that the classifier or vectorizer takes can be tuned!

# The pipeline notation in scikit learn

You might have noticed the Pipeline construct in the last example.

- Machine learning involves multiple steps (e.g., preprocessing $\rightarrow$ vectorizer $\rightarrow$ classification)
- We did all of them seperately before
- Nothing wrong with that, but to ease use and evaluation *of the whole process* (as needed for the gridsearch), we can define a `pipeline`.

# Example with a pipeline (and add cross-validation)

```
1  from sklearn.feature_extraction.text import TfidfVectorizer
2  from sklearn.linear_model import LogisticRegressionCV
3  from sklearn.pipeline import make_pipeline
4
5  vec = TfidfVectorizer()
6  clf = LogisticRegressionCV()
7  pipe = make_pipeline(vec, clf)
8
9  pipe.fit([r[0] for r in reviews], [r[1] for r in reviews])
10 predictions = pipe.predict([r[0] for r in test])
```

# Pipeline takeaway

- In principle, just a different way to write what we already did
- The more steps, the more relevant (e.g., preprocessing $\rightarrow$ vectorizer $\rightarrow$ dimensionality-reduction $\rightarrow$ classification)
- The more you rely on automated evaluation (e.g., grid search) of *multiple* steps in the pipeline, the more useful it is

# Grid-search takeaway

- When you want to systematically test what happens when you vary a hyperparameter, use grid-search to automatically do so and select the best value
- sometimes already implemented (e.g., `LogisticRegressionCV` as direct replacement for `LogisticRegression`)
- But `GridSearchCV` is very flexible: can be used in combination with pipeline for very different purposes

# Grid-search takeaway

- When you want to systematically test what happens when you vary a hyperparameter, use grid-search to automatically do so and select the best value
- sometimes already implemented (e.g., `LogisticRegressionCV` as direct replacement for `LogisticRegression`)
- But `GridSearchCV` is very flexible: can be used in combination with pipeline for very different purposes

# Grid-search takeaway

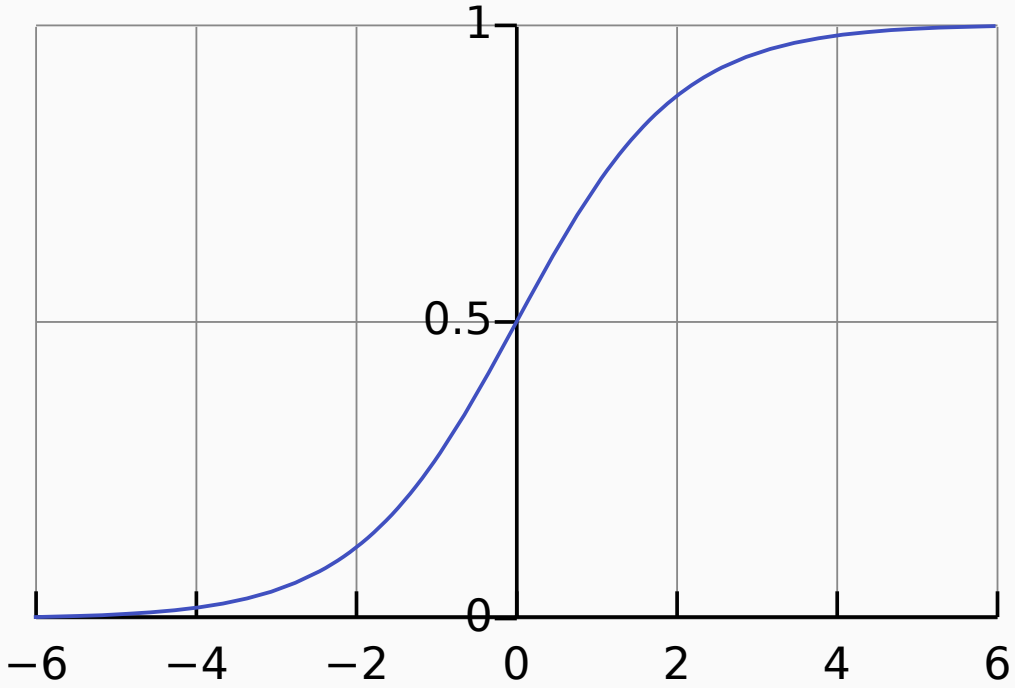
- When you want to systematically test what happens when you vary a hyperparameter, use grid-search to automatically do so and select the best value
- sometimes already implemented (e.g., `LogisticRegressionCV` as direct replacement for `LogisticRegression`)
- But `GridSearchCV` is very flexible: can be used in combination with pipeline for very different purposes

# More suggestions

# More suggestions

## Tuning decision thresholds with ROC curves

# From estimate to label




In logistic regression, we use the *sigmoid function* to transform the estimates into probabilities.

To transform the probabilities into binary labels, we use a cutoff (default: 0.5).

# Why use 0.5 as cutoff?

- It makes most sense (intuitively, mathematically)
- But remember our precision/recall tradeoff: maybe we want to be 'stricter' or 'less strict'
- Maybe it is importance to us that our classifier is balanced and equally good in predicting both classes, even if overall accuracy suffers (slightly)

Let's see what happens if we plot False Positives against True Positives (ROC-curve)

# Why use 0.5 as cutoff?

- It makes most sense (intuitively, mathematically)
- But remember our precision/recall tradeoff: maybe we want to be 'stricter' or 'less strict'
- Maybe it is importance to us that our classifier is balanced and equally good in predicting both classes, even if overall accuracy suffers (slightly)

Let's see what happens if we plot False Positives against True Positives (ROC-curve)

# Why use 0.5 as cutoff?

- It makes most sense (intuitively, mathematically)
- But remember our precision/recall tradeoff: maybe we want to be 'stricter' or 'less strict'
- Maybe it is importance to us that our classifier is balanced and equally good in predicting both classes, even if overall accuracy suffers (slightly)

Let's see what happens if we plot False Positives against True Positives (ROC-curve)

# Why use 0.5 as cutoff?

- It makes most sense (intuitively, mathematically)
- But remember our precision/recall tradeoff: maybe we want to be 'stricter' or 'less strict'
- Maybe it is importance to us that our classifier is balanced and equally good in predicting both classes, even if overall accuracy suffers (slightly)

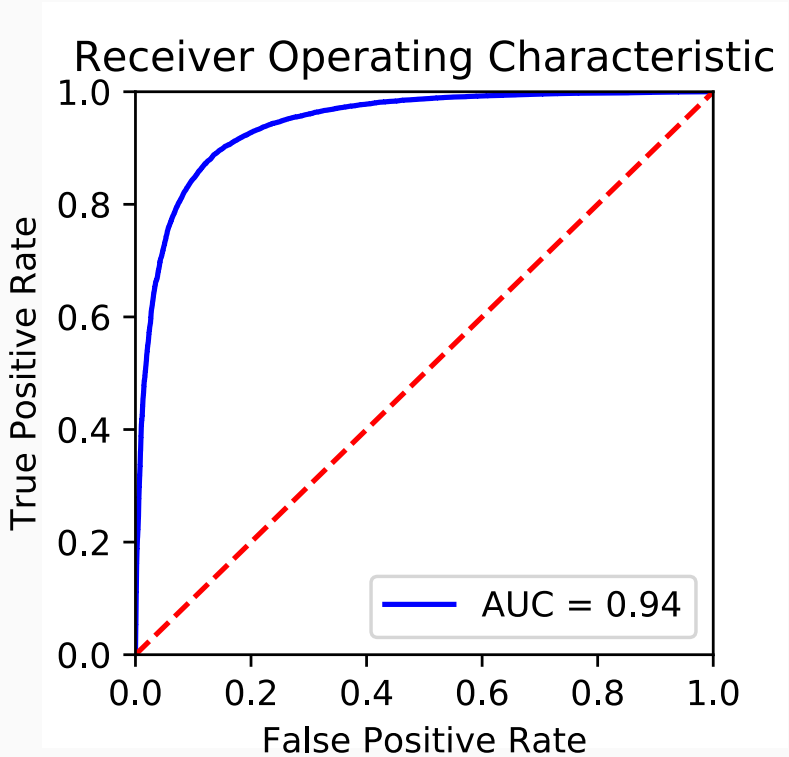
Let's see what happens if we plot False Positives against True Positives (ROC-curve)

# Why use 0.5 as cutoff?

- It makes most sense (intuitively, mathematically)
- But remember our precision/recall tradeoff: maybe we want to be 'stricter' or 'less strict'
- Maybe it is importance to us that our classifier is balanced and equally good in predicting both classes, even if overall accuracy suffers (slightly)

Let's see what happens if we plot False Positives against True Positives (ROC-curve)

# ROC Curve




- If we choose a threshold such that we get very little false positives, we also get too little true positives.
- Optimum in the upper left corner

# So, how to we determine the exact value?

See notebook

https://github.com/damian0604/bdaca/blob/master/12ec/week10/Determining%20the%20cutoff-point%20in%20logistic%20regression.ipynb

# Some further ideas to look into

## Balancing classes

Your classifier probably works better if you have approximately the same amount of annotated training data for both classes (e.g., pos/neg). If getting such data is not an option, you may consider weighing accordingly, e.g. using

```
LogisticRegression(class_weight='balanced')
```

# Some further ideas to look into

## More advanced pipelines

Consider constructing advanced pipelines, including a dimension reduction step:

https://scikit-learn.org/stable/auto_examples/compose/plot_digits_pipe.html

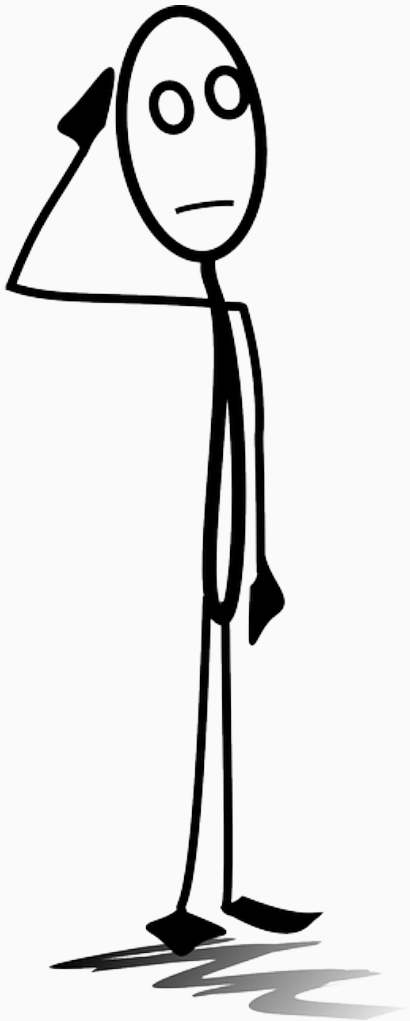
# Some further ideas to look into

**Combine different feature sets**

E.g, use BOW-features as well as features such as sentence length, number of sentences (or whatever)

https://scikit-learn.org/stable/auto_examples/hetero_feature_union.html or see example in repository

# Exercise



*Any questions?*

# Let's exercise!