5
Classification – Detecting Poor Answers

Now that we are able to extract useful features from text, we can take on the challenge of building a classifier using real data. Let's come back to our imaginary website in *Chapter 3*, *Clustering – Finding Related Posts*, where users can submit questions and get them answered.

A continuous challenge for owners of those Q&A sites is to maintain a decent level of quality in the posted content. Sites such as StackOverflow make considerable efforts to encourage users with diverse possibilities to score content and offer badges and bonus points in order to encourage the users to spend more energy on carving out the question or crafting a possible answer.

One particular successful incentive is the ability for the asker to flag one answer to their question as the accepted answer (again there are incentives for the asker to flag answers as such). This will result in more score points for the author of the flagged answer.

Would it not be very useful to the user to immediately see how good his answer is while he is typing it in? That means, the website would continuously evaluate his work-in-progress answer and provide feedback as to whether the answer shows some signs of a poor one. This will encourage the user to put more effort into writing the answer (providing a code example? including an image?), and thus improve the overall system.

Let's build such a mechanism in this chapter.

Sketching our roadmap

As we will build a system using real data that is very noisy, this chapter is not for the fainthearted, as we will not arrive at the golden solution of a classifier that achieves 100 percent accuracy; often, even humans disagree whether an answer was good or not (just look at some of the StackOverflow comments). Quite the contrary, we will find out that some problems like this one are so hard that we have to adjust our initial goals on the way. But on the way, we will start with the nearest neighbor approach, find out why it is not very good for the task, switch over to logistic regression, and arrive at a solution that will achieve good enough prediction quality, but on a smaller part of the answers. Finally, we will spend some time looking at how to extract the winner to deploy it on the target system.

Learning to classify classy answers

In classification, we want to find the corresponding **classes**, sometimes also called **labels**, for given data instances. To be able to achieve this, we need to answer two questions:

- How should we represent the data instances?
- Which model or structure should our classifier possess?

Tuning the instance

In its simplest form, in our case, the data instance is the text of the answer and the label would be a binary value indicating whether the asker accepted this text as an answer or not. Raw text, however, is a very inconvenient representation to process for most machine learning algorithms. They want numbers. And it will be our task to extract useful features from the raw text, which the machine learning algorithm can then use to learn the right label for it.

Tuning the classifier

Once we have found or collected enough (text, label) pairs, we can train a **classifier**. For the underlying structure of the classifier, we have a wide range of possibilities, each of them having advantages and drawbacks. Just to name some of the more prominent choices, there are logistic regression, decision trees, SVMs, and Naïve Bayes. In this chapter, we will contrast the instance-based method from the last chapter, nearest neighbor, with model-based logistic regression.

Fetching the data

Luckily for us, the team behind StackOverflow provides most of the data behind the StackExchange universe to which StackOverflow belongs under a cc-wiki license. At the time of writing this book, the latest data dump can be found at `https://archive.org/details/stackexchange`. It contains data dumps of all Q&A sites of the StackExchange family. For StackOverflow, you will find multiple files, of which we only need the `stackoverflow.com-Posts.7z` file, which is 5.2 GB.

After downloading and extracting it, we have around 26 GB of data in the format of XML, containing all questions and answers as individual `row` tags within the `root` tag posts:

```
<?xml version="1.0" encoding="utf-8"?>
<posts>
...
  <row Id="4572748" PostTypeId="2" ParentId="4568987"
CreationDate="2011-01-01T00:01:03.387" Score="4" ViewCount=""
Body="&lt;p&gt;IANAL, but &lt;a
href=&quot;http://support.apple.com/kb/HT2931&quot;
rel=&quot;nofollow&quot;&gt;this&lt;/a&gt; indicates to me that you
cannot use the loops in your
application:&lt;/p&gt;&#xA;&#xA;&lt;blockquote&gt;&#xA;
&lt;p&gt;...however, individual audio loops may&#xA;  not be
commercially or otherwise&#xA;  distributed on a standalone basis,
nor&#xA;  may they be repackaged in whole or in&#xA;  part as audio
samples, sound effects&#xA;  or music beds.&quot;&lt;/p&gt;&#xA;
&#xA;  &lt;p&gt;So don't worry, you can make&#xA;  commercial music
with GarageBand, you&#xA;  just can't distribute the loops as&#xA;
loops.&lt;/p&gt;&#xA;&lt;/blockquote&gt;&#xA;" OwnerUserId="203568"
LastActivityDate="2011-01-01T00:01:03.387" CommentCount="1" />
…
</posts>
```

Name	Type	Description
`Id`	`Integer`	This is a unique identifier.
`PostTypeId`	`Integer`	This describes the category of the post. The values interesting to us are the following: • Question • Answer Other values will be ignored.
`ParentId`	`Integer`	This is a unique identifier of the question to which this answer belongs (missing for questions).

Name	Type	Description
`CreationDate`	`DateTime`	This is the date of submission.
`Score`	`Integer`	This is the score of the post.
`ViewCount`	`Integer or empty`	This is the number of user views for this post.
`Body`	`String`	This is the complete post as encoded HTML text.
`OwnerUserId`	`Id`	This is a unique identifier of the poster. If 1, then it is a wiki question.
`Title`	`String`	This is the title of the question (missing for answers).
`AcceptedAnswerId`	`Id`	This is the ID for the accepted answer (missing for answers).
`CommentCount`	`Integer`	This is the number of comments for the post.

Slimming the data down to chewable chunks

To speed up our experimentation phase, we should not try to evaluate our classification ideas on the huge XML file. Instead, we should think of how we could trim it down so that we still keep a representable snapshot of it while being able to quickly test our ideas. If we filter the XML for `row` tags that have a creation date of, for example, 2012, we still end up with over 6 million posts (2,323,184 questions and 4,055,999 answers), which should be enough to pick our training data from for now. We also do not want to operate on the XML format as it will slow us down, too. The simpler the format, the better. That's why we parse the remaining XML using Python's `cElementTree` and write it out to a tab-separated file.

Preselection and processing of attributes

To cut down the data even more, we can certainly drop attributes that we think will not help the classifier in distinguishing between good and not-so-good answers. But we have to be cautious here. Although some features are not directly impacting the classification, they are still necessary to keep.

The `PostTypeId` attribute, for example, is necessary to distinguish between questions and answers. It will not be picked to serve as a feature, but we will need it to filter the data.

`CreationDate` could be interesting to determine the time span between posting the question and posting the individual answers, so we keep it. The `Score` is of course important as an indicator for the community's evaluation.

ViewCount, in contrast, is most likely of no use for our task. Even if it would help the classifier to distinguish between good and bad, we would not have this information at the time when an answer is being submitted. Drop it!

The Body attribute obviously contains the most important information. As it is encoded HTML, we will have to decode to plain text.

OwnerUserId is only useful if we take user-dependent features in to account, which we won't. Although we drop it here, we encourage you to use it to build a better classifier (maybe in connection with stackoverflow.com-Users.7z).

The Title attribute is also ignored here, although it could add some more information about the question.

CommentCount is also ignored. Similar to ViewCount, it could help the classifier with posts that are out there for a while (more comments = more ambiguous post?). It will, however, not help the classifier at the time an answer is posted.

AcceptedAnswerId is similar to Score in that it is an indicator of a post's quality. As we will access this per answer, instead of keeping this attribute, we will create the new attribute IsAccepted, which is 0 or 1 for answers and ignored for questions (ParentId=-1).

We end up with the following format:

```
Id <TAB> ParentId <TAB> IsAccepted <TAB> TimeToAnswer <TAB> Score
<TAB> Text
```

For the concrete parsing details, please refer to so_xml_to_tsv.py and choose_instance.py. Suffice to say that in order to speed up processing, we will split the data into two files: in meta.json, we store a dictionary mapping a post's Id value to its other data except Text in JSON format so that we can read it in the proper format. For example, the score of a post would reside at meta[Id]['Score']. In data.tsv, we store the Id and Text values, which we can easily read with the following method:

```
def fetch_posts():
    for line in open("data.tsv", "r"):
        post_id, text = line.split("\t")
        yield int(post_id), text.strip()
```

Defining what is a good answer

Before we can train a classifier to distinguish between good and bad answers, we have to create the training data. So far, we only have a bunch of data. What we still have to do is define labels.

We could, of course, simply use the `IsAccepted` attribute as a label. After all, that marks the answer that answered the question. However, that is only the opinion of the asker. Naturally, the asker wants to have a quick answer and accepts the first *best* answer. If over time more answers are submitted, some of them will tend to be better than the already accepted one. The asker, however, seldom gets back to the question and changes his mind. So we end up with many questions that have accepted answers that are not scored highest.

At the other extreme, we could simply always take the best and worst scored answer per question as positive and negative examples. However, what do we do with questions that have only good answers, say, one with two and the other with four points? Should we really take an answer with, for example, two points as a negative example just because it happened to be the one with the lower score?

We should settle somewhere between these extremes. If we take all answers that are scored higher than zero as positive and all answers with zero or less points as negative, we end up with quite reasonable labels:

```
>>> all_answers = [q for q,v in meta.items() if v['ParentId']!=-1]
>>> Y = np.asarray([meta[answerId]['Score']>0 for answerId in
all_answers])
```

Creating our first classifier

Let's start with the simple and beautiful nearest neighbor method from the previous chapter. Although it is not as advanced as other methods, it is very powerful: as it is not model-based, it can *learn* nearly any data. But this beauty comes with a clear disadvantage, which we will find out very soon.

Starting with kNN

This time, we won't implement it ourselves, but rather take it from the `sklearn` toolkit. There, the classifier resides in `sklearn.neighbors`. Let's start with a simple 2-Nearest Neighbor classifier:

```
>>> from sklearn import neighbors
>>> knn = neighbors.KNeighborsClassifier(n_neighbors=2)
>>> print(knn)
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30,
metric='minkowski', n_neighbors=2, p=2, weights='uniform')
```

It provides the same interface as all other estimators in `sklearn`: we train it using `fit()`, after which we can predict the class of new data instances using `predict()`:

```
>>> knn.fit([[1],[2],[3],[4],[5],[6]], [0,0,0,1,1,1])
>>> knn.predict(1.5)
array([0])
>>> knn.predict(37)
array([1])
>>> knn.predict(3)
array([0])
```

To get the class probabilities, we can use `predict_proba()`. In this case of having two classes, `0` and `1`, it will return an array of two elements:

```
>>> knn.predict_proba(1.5)
array([[ 1.,  0.]])
>>> knn.predict_proba(37)
array([[ 0.,  1.]])
>>> knn.predict_proba(3.5)
array([[ 0.5,  0.5]])
```

Engineering the features

So, what kind of features can we provide to our classifier? What do we think will have the most discriminative power?

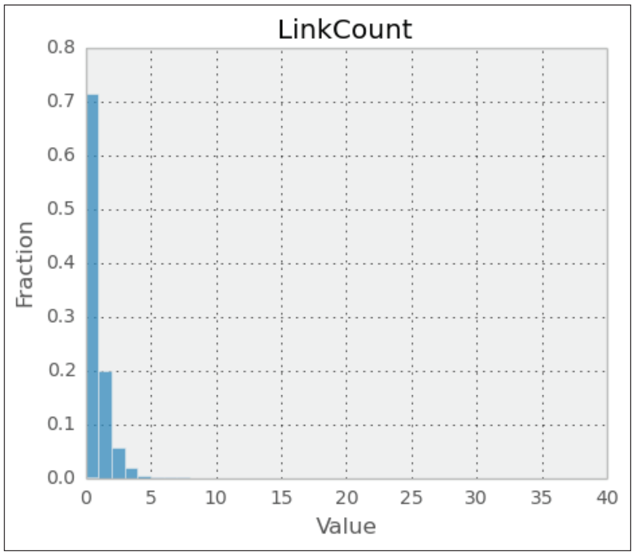
`TimeToAnswer` is already there in our `meta` dictionary, but it probably won't provide much value on its own. Then there is only `Text`, but in its raw form, we cannot pass it to the classifier, as the features must be in numerical form. We will have to do the dirty (and fun!) work of extracting features from it.

What we could do is check the number of HTML links in the answer as a proxy for quality. Our hypothesis would be that more hyperlinks in an answer indicate better answers and thus a higher likelihood of being up-voted. Of course, we want to only count links in normal text and not code examples:

```
import re
code_match = re.compile('<pre>(.*?)</pre>',
                        re.MULTILINE | re.DOTALL)
link_match = re.compile('<a href="http://.*?".*?>(.*?)</a>',
                        re.MULTILINE | re.DOTALL)
tag_match = re.compile('<[^>]*>',
                        re.MULTILINE | re.DOTALL)


def extract_features_from_body(s):
    link_count_in_code = 0
    # count links in code to later subtract them
    for match_str in code_match.findall(s):
        link_count_in_code += len(link_match.findall(match_str))

    return len(link_match.findall(s)) - link_count_in_code
```

For production systems, we would not want to parse HTML content with regular expressions. Instead, we should rely on excellent libraries such as BeautifulSoup, which does a marvelous job of robustly handling all the weird things that typically occur in everyday HTML.

With this in place, we can generate one feature per answer. But before we train the classifier, let's first have a look at what we will train it with. We can get a first impression with the frequency distribution of our new feature. This can be done by plotting the percentage of how often each value occurs in the data. Have a look at the following plot:



With the majority of posts having no link at all, we know now that this feature will not make a good classifier alone. Let's nevertheless try it out to get a first estimation of where we are.

Training the classifier

We have to pass the feature array together with the previously defined labels `Y` to the kNN learner to obtain a classifier:

```
X = np.asarray([extract_features_from_body(text) for post_id, text in
                fetch_posts() if post_id in all_answers])
knn = neighbors.KNeighborsClassifier()
knn.fit(X, Y)
```

Using the standard parameters, we just fitted a 5NN (meaning NN with `k=5`) to our data. Why 5NN? Well, at the current state of our knowledge about the data, we really have no clue what the right `k` should be. Once we have more insight, we will have a better idea of how to set `k`.

Measuring the classifier's performance

We have to be clear about what we want to measure. The naïve but easiest way is to simply calculate the average prediction quality over the test set. This will result in a value between 0 for predicting everything wrongly and 1 for perfect prediction. The accuracy can be obtained through `knn.score()`.

But as we learned in the previous chapter, we will not do it just once, but apply cross-validation here using the readymade `KFold` class from `sklearn.cross_validation`. Finally, we will then average the scores on the test set of each fold and see how much it varies using standard deviation:

```
from sklearn.cross_validation import KFold
scores = []

cv = KFold(n=len(X), k=10, indices=True)

for train, test in cv:
    X_train, y_train = X[train], Y[train]
    X_test, y_test = X[test], Y[test]
    clf = neighbors.KNeighborsClassifier()
    clf.fit(X, Y)
    scores.append(clf.score(X_test, y_test))


print("Mean(scores)=%.5f\tStddev(scores)=%.5f"\
      %(np.mean(scores), np.std(scores)))
```

Here is the output:

```
Mean(scores)=0.50250    Stddev(scores)=0.055591
```

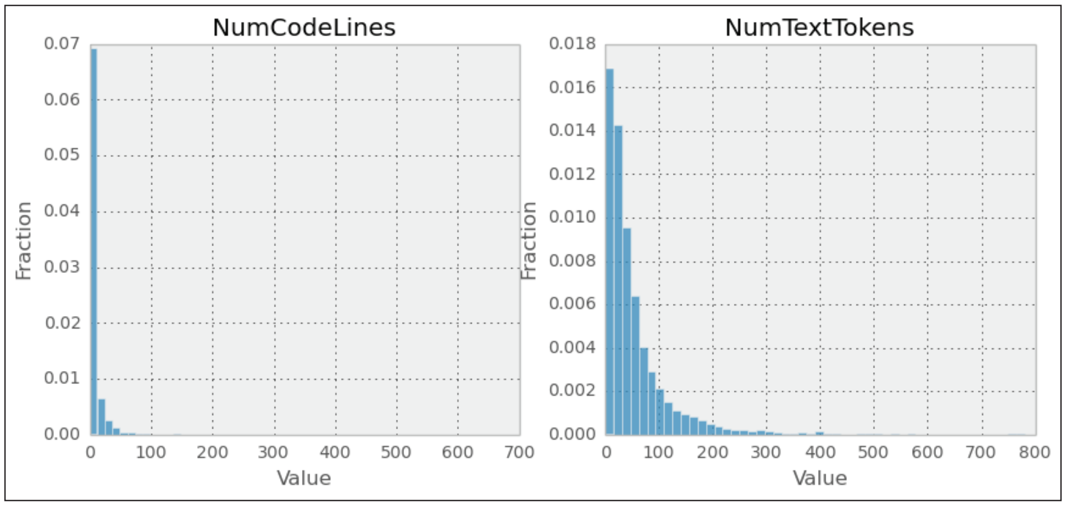
Now that is far from being usable. With only 55 percent accuracy, it is not much better than tossing a coin. Apparently, the number of links in a post is not a very good indicator for the quality of a post. So, we can say that this feature does not have much discriminative power—at least not for kNN with `k=5`.

Designing more features

In addition to using the number of hyperlinks as a proxy for a post's quality, the number of code lines is possibly another good one, too. At least it is a good indicator that the post's author is interested in answering the question. We can find the code embedded in the `<pre>…</pre>` tag. And once we have it extracted, we should count the number of words in the post while ignoring code lines:

```
def extract_features_from_body(s):
    num_code_lines = 0
    link_count_in_code = 0
    code_free_s = s

    # remove source code and count how many lines
    for match_str in code_match.findall(s):
        num_code_lines += match_str.count('\n')
        code_free_s = code_match.sub("", code_free_s)

        # Sometimes source code contains links,
        # which we don't want to count
        link_count_in_code += len(link_match.findall(match_str))

    links = link_match.findall(s)
    link_count = len(links)
    link_count -= link_count_in_code
    html_free_s = re.sub(" +", " ",
          tag_match.sub('',  code_free_s)).replace("\n", "")
    link_free_s = html_free_s

    # remove links from text before counting words
    for link in links:
        if link.lower().startswith("http://"):
            link_free_s = link_free_s.replace(link,'')

    num_text_tokens = html_free_s.count(" ")

    return num_text_tokens, num_code_lines, link_count
```

Looking at them, we notice that at least the number of words in a post shows higher variability:



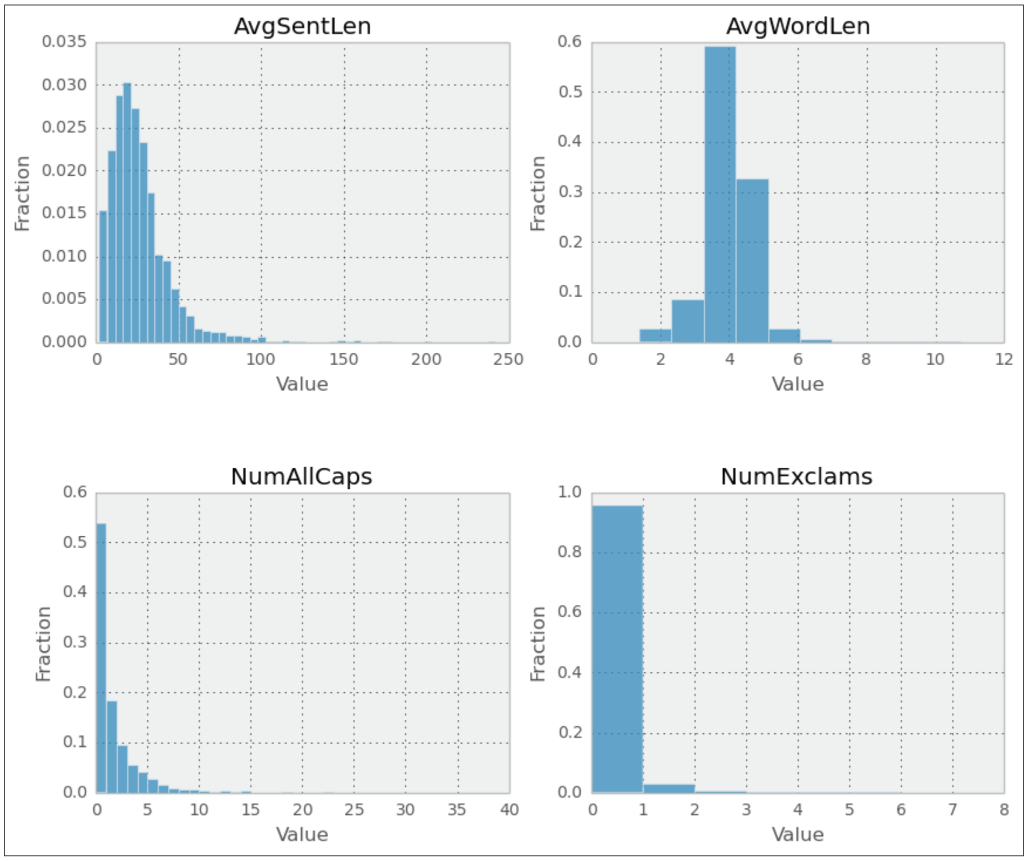
Training on the bigger feature space improves accuracy quite a bit:

```
Mean(scores)=0.59800    Stddev(scores)=0.02600
```

But still, this would mean that we would classify roughly 4 out of 10 wrong. At least we are going in the right direction. More features lead to higher accuracy, which leads us to adding more features. Therefore, let's extend the feature space by even more features:

- `AvgSentLen`: This measures the average number of words in a sentence. Maybe there is a pattern that particularly good posts don't overload the reader's brain with overly long sentences?
- `AvgWordLen`: Similar to `AvgSentLen`, this feature measures the average number of characters in the words of a post.
- `NumAllCaps`: This measures the number of words that are written in uppercase, which is considered bad style.
- `NumExclams`: This measures the number of exclamation marks.

The following charts show the value distributions for average sentence and word lengths and number of uppercase words and exclamation marks:



With these four additional features, we now have seven features representing the individual posts. Let's see how we progress:

```
Mean(scores)=0.61400    Stddev(scores)= 0.02154
```

Now, that's interesting. We added four more features and don't get anything in return. How can that be?

To understand this, we have to remind ourselves how kNN works. Our 5NN classifier determines the class of a new post by calculating the seven aforementioned features, `LinkCount`, `NumTextTokens`, `NumCodeLines`, `AvgSentLen`, `AvgWordLen`, `NumAllCaps`, and `NumExclams`, and then finds the five nearest other posts. The new post's class is then the majority of the classes of those nearest posts. The nearest posts are determined by calculating the Euclidean distance (as we did not specify it, the classifier was initialized with the default `p=2`, which is the parameter in the Minkowski distance). That means that all seven features are treated similarly. kNN does not really learn that, for instance, `NumTextTokens` is good to have but much less important than `NumLinks`. Let's consider the following two posts A and B that only differ in the following features and how they compare to a new post:

Post	**NumLinks**	**NumTextTokens**
A	2	20
B	0	25
new	1	23

Although we would think that links provide more value than mere text, post B would be considered more similar to the new post than post A.

Clearly, kNN has a hard time in correctly using the available data.

Deciding how to improve

To improve on this, we basically have the following options:

- **Add more data**: Maybe it is just not enough data for the learning algorithm and we should simply add more training data?
- **Play with the model complexity**: Maybe the model is not complex enough? Or maybe it is already too complex? In this case, we could decrease *k* so that it would take less nearest neighbors into account and thus be better in predicting non-smooth data. Or we could increase it to achieve the opposite.

- **Modify the feature space**: Maybe we do not have the right set of features? We could, for example, change the scale of our current features or design even more new features. Or should we rather remove some of our current features in case some features are aliasing others?
- **Change the model**: Maybe kNN is in general not a good fit for our use case such that it will never be capable of achieving good prediction performance, no matter how complex we allow it to be and how sophisticated the feature space will become?

In real life, at this point, people often try to improve the current performance by randomly picking one of the these options and trying them out in no particular order, hoping to find the golden configuration by chance. We could do the same here, but it will surely take longer than making informed decisions. Let's take the informed route, for which we need to introduce the bias-variance tradeoff.

Bias-variance and their tradeoff

In *Chapter 1*, *Getting Started with Python Machine Learning*, we tried to fit polynomials of different complexities controlled by the dimensionality parameter `d` to fit the data. We realized that a two-dimensional polynomial, a straight line, does not fit the example data very well, because the data was not of linear nature. No matter how elaborate our fitting procedure would have been, our two-dimensional model would see everything as a straight line. We say that it is too biased for the data at hand. It is under-fitting.

We played a bit with the dimensions and found out that the 100-dimensional polynomial is actually fitting very well to the data on which it was trained (we did not know about train-test splits at that time). However, we quickly found out that it was fitting too well. We realized that it was over-fitting so badly, that with different samples of the data points, we would have gotten totally different 100-dimensional polynomials. We say that the model has a too high variance for the given data, or that it is over-fitting.

These are the extremes between which most of our machine learning problems reside. Ideally, we want to have both, low bias and low variance. But, we are in a bad world, and have to tradeoff between them. If we improve on one, we will likely get worse on the other.

Fixing high bias

Let's now assume we suffer from high bias. In that case, adding more training data clearly does not help. Also, removing features surely will not help, as our model would have already been overly simplistic.

The only possibilities we have in this case are to get more features, make the model more complex, or change the model.

Fixing high variance

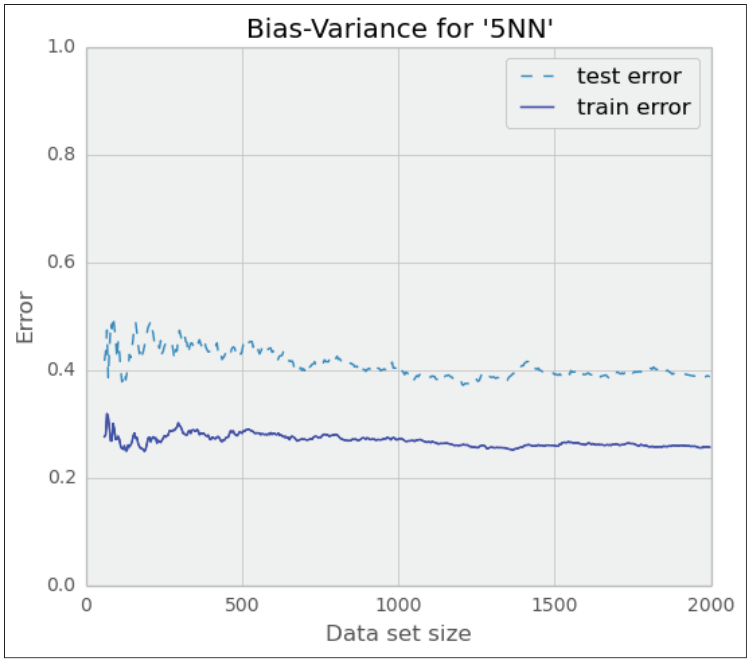
If, on the contrary, we suffer from high variance, that means that our model is too complex for the data. In this case, we can only try to get more data or decrease the complexity. This would mean to increase *k* so that more neighbors would be taken into account or to remove some of the features.

High bias or low bias

To find out what our problem actually is, we have to simply plot the train and test errors over the data size.

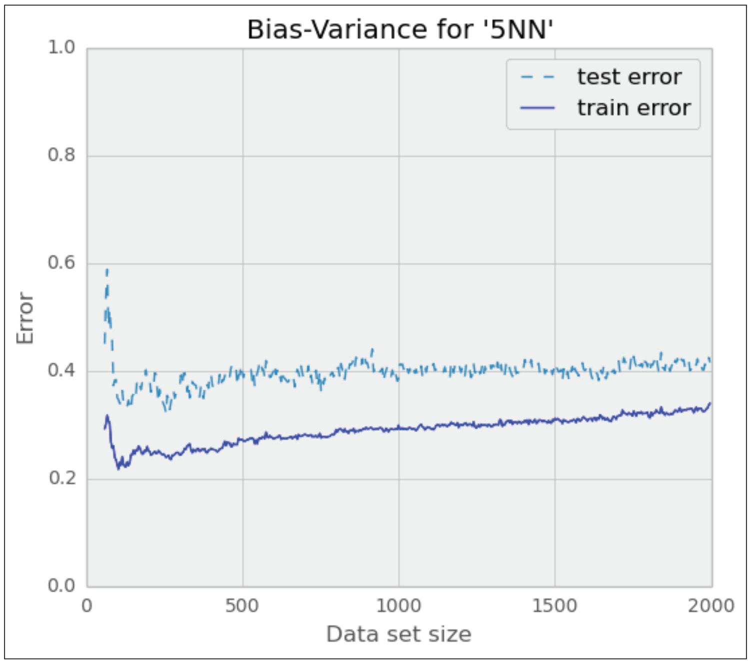
High bias is typically revealed by the test error decreasing a bit at the beginning, but then settling at a very high value with the train error approaching with a growing dataset size. High variance is recognized by a big gap between both curves.

Plotting the errors for different dataset sizes for 5NN shows a big gap between train and test errors, hinting at a high variance problem:



Looking at the graph, we immediately see that adding more training data will not help, as the dashed line corresponding to the test error seems to stay above 0.4. The only option we have is to decrease the complexity, either by increasing *k* or by reducing the feature space.

Reducing the feature space does not help here. We can easily confirm this by plotting the graph for a simplified feature space of only `LinkCount` and `NumTextTokens`:

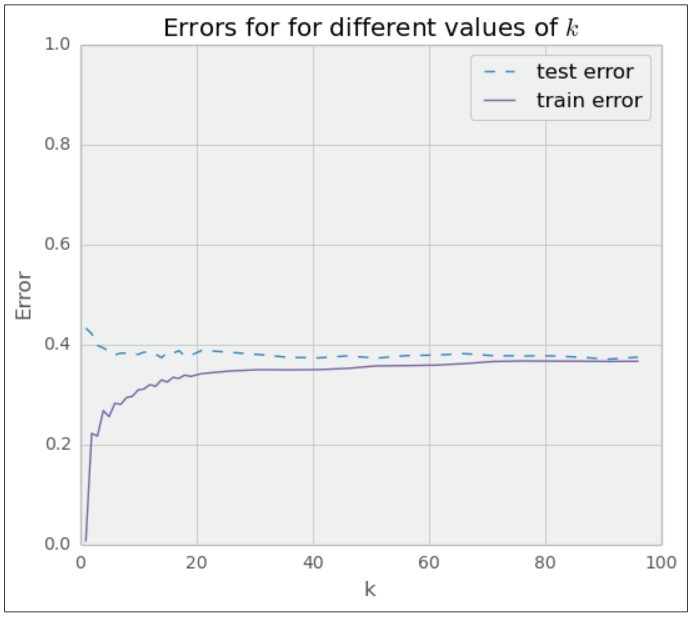


We get similar graphs for other smaller feature sets. No matter what subset of features we take, the graph would look similar.

At least reducing the model complexity by increasing *k* shows some positive impact:

k	**mean(scores)**	**stddev(scores)**
40	0.62800	0.03750
10	0.62000	0.04111
5	0.61400	0.02154

But it is not enough, and also comes at a price of lower classification runtime performance. Take, for instance, `k=40`, where we have a very low test error. To classify a new post, we would need to find the 40 nearest other posts to decide whether the new post is a good one or not:



Clearly, it seems to be an issue with using nearest neighbor for our scenario. And it has another real disadvantage. Over time, we will get more and more posts into our system. As the nearest neighbor method is an instance-based approach, we will have to store all posts in our system. The more we get, the slower the prediction will be. This is different with model-based approaches, where one tries to derive a model from the data.

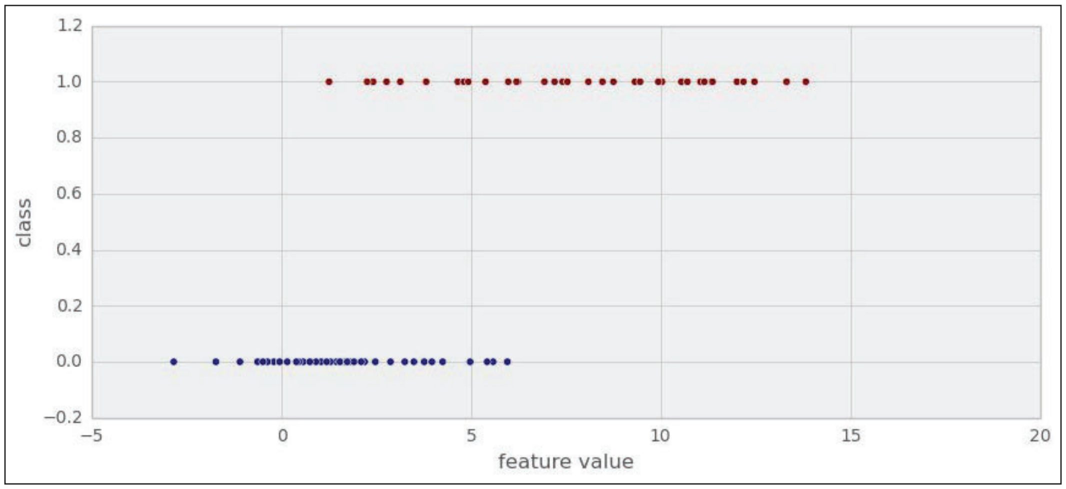
There we are, with enough reasons now to abandon the nearest neighbor approach to look for better places in the classification world. Of course, we will never know whether there is the one golden feature we just did not happen to think of. But for now, let's move on to another classification method that is known to work great in text-based classification scenarios.

Using logistic regression

Contrary to its name, logistic regression is a classification method. It is a very powerful one when it comes to text-based classification; it achieves this by first doing a regression on a logistic function, hence the name.

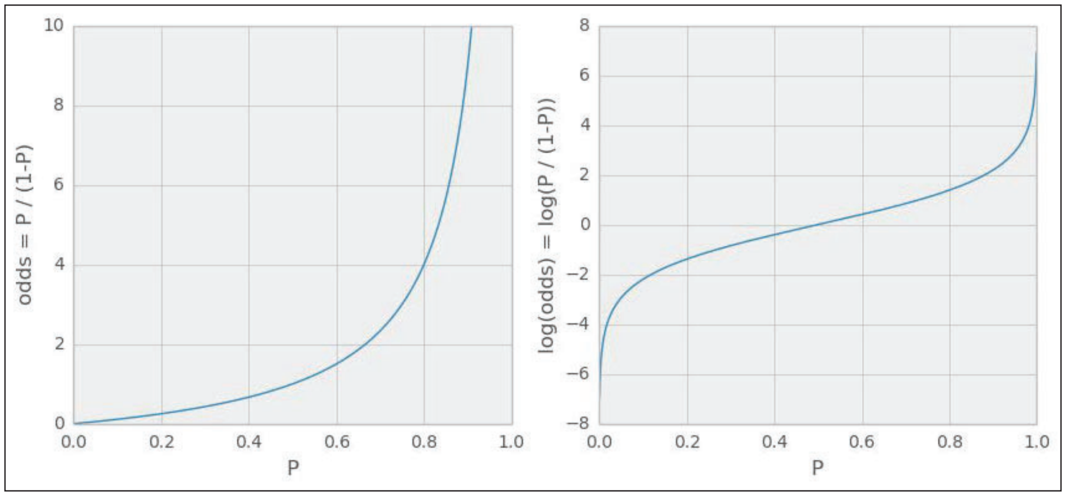
A bit of math with a small example

To get an initial understanding of the way logistic regression works, let's first take a look at the following example where we have artificial feature values *X* plotted with the corresponding classes, 0 or 1. As we can see, the data is noisy such that classes overlap in the feature value range between 1 and 6. Therefore, it is better to not directly model the discrete classes, but rather the probability that a feature value belongs to class 1, *P(X)*. Once we possess such a model, we could then predict class 1 if *P(X)>0.5*, and class 0 otherwise.



Mathematically, it is always difficult to model something that has a finite range, as is the case here with our discrete labels 0 and 1. We can, however, tweak the probabilities a bit so that they always stay between 0 and 1. And for that, we will need the odds ratio and the logarithm of it.

Let's say a feature has the probability of 0.9 that it belongs to class 1, *P(y=1) = 0.9*. The odds ratio is then *P(y=1)/P(y=0) = 0.9/0.1 = 9*. We could say that the chance is 9:1 that this feature maps to class 1. If *P(y=0.5)*, we would consequently have a 1:1 chance that the instance is of class 1. The odds ratio is bounded by 0, but goes to infinity (the left graph in the following set of graphs). If we now take the logarithm of it, we can map all probabilities between 0 and 1 to the full range from negative to positive infinity (the right graph in the following set of graphs). The nice thing is that we still maintain the relationship that higher probability leads to a higher log of odds, just not limited to 0 and 1 anymore.



This means that we can now fit linear combinations of our features (OK, we only have one and a constant, but that will change soon) to the $\log(odds)$ values. In a sense, we replace the linear from *Chapter 1*, *Getting Started with Python Machine Learning*, $y_i = c_0 + c_1 x_i$ with $log\left(\frac{p_i}{1-p_i}\right) = c_0 + c_1 x$ (replacing *y* with *log(odds)*).

We can solve this for p_i, so that we have $p_i = \frac{1}{1+e^{-(c_0+c_1 x_i)}}$.

We simply have to find the right coefficients, such that the formula gives the lowest errors for all our (x_i, p_i) pairs in our data set, but that will be done by scikit-learn.

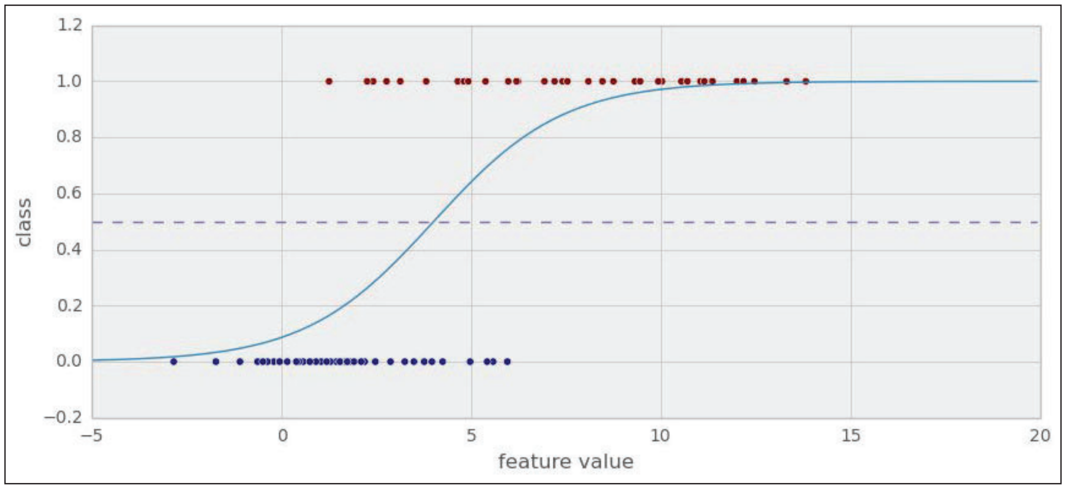
After fitting, the formula will give the probability for every new data point *x* that belongs to class 1:

```
>>> from sklearn.linear_model import LogisticRegression
>>> clf = LogisticRegression()
>>> print(clf)
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False,
fit_intercept=True, intercept_scaling=1, penalty=l2, tol=0.0001)
>>> clf.fit(X, y)
>>> print(np.exp(clf.intercept_), np.exp(clf.coef_.ravel()))
[ 0.09437188] [ 1.80094112]
>>> def lr_model(clf, X):
...     return 1 / (1 + np.exp(-(clf.intercept_ + clf.coef_*X)))
>>> print("P(x=-1)=%.2f\tP(x=7)=%.2f"%(lr_model(clf, -1),
lr_model(clf, 7)))
P(x=-1)=0.05    P(x=7)=0.85
```

You might have noticed that scikit-learn exposes the first coefficient through the special field `intercept_`.

If we plot the fitted model, we see that it makes perfect sense given the data:



Applying logistic regression to our post classification problem

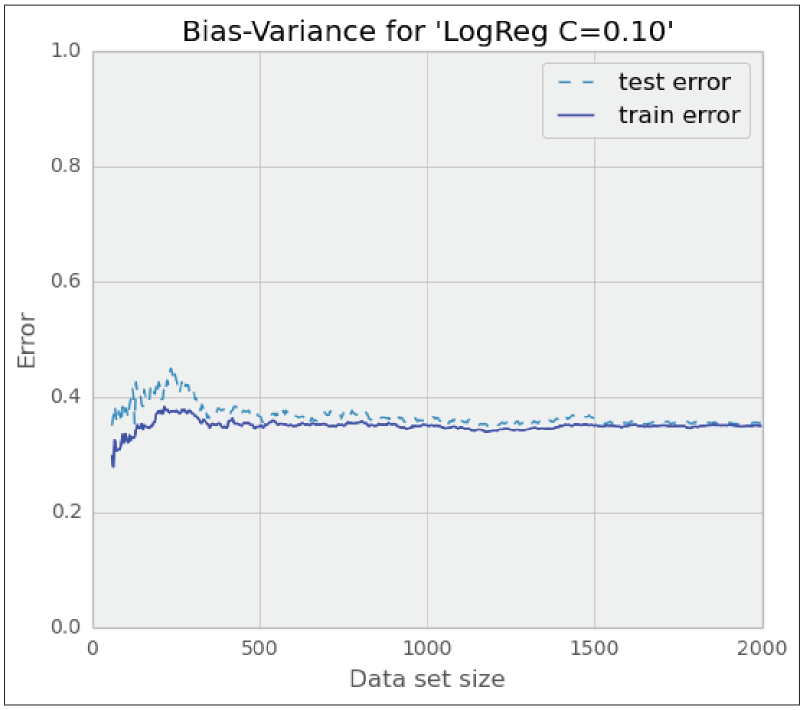
Admittedly, the example in the previous section was created to show the beauty of logistic regression. How does it perform on the real, noisy data?

Comparing it to the best nearest neighbor classifier (`k=40`) as a baseline, we see that it performs a bit better, but also won't change the situation a whole lot.

Method	**mean(scores)**	**stddev(scores)**
LogReg C=0.1	0.64650	0.03139
LogReg C=1.00	0.64650	0.03155
LogReg C=10.00	0.64550	0.03102
LogReg C=0.01	0.63850	0.01950
40NN	0.62800	0.03750

We have shown the accuracy for different values of the regularization parameter `C`. With it, we can control the model complexity, similar to the parameter `k` for the nearest neighbor method. Smaller values for `C` result in more penalization of the model complexity.

A quick look at the bias-variance chart for one of our best candidates, `C=0.1`, shows that our model has high bias—test and train error curves approach closely but stay at unacceptable high values. This indicates that logistic regression with the current feature space is under-fitting and cannot learn a model that captures the data correctly:



So what now? We switched the model and tuned it as much as we could with our current state of knowledge, but we still have no acceptable classifier.

More and more it seems that either the data is too noisy for this task or that our set of features is still not appropriate to discriminate the classes well enough.

Looking behind accuracy – precision and recall

Let's step back and think again about what we are trying to achieve here. Actually, we do not need a classifier that perfectly predicts good and bad answers as we measured it until now using accuracy. If we can tune the classifier to be particularly good at predicting one class, we could adapt the feedback to the user accordingly. If we, for example, had a classifier that was always right when it predicted an answer to be bad, we would give no feedback until the classifier detected the answer to be bad. On the contrary, if the classifier exceeded in predicting answers to be good, we could show helpful comments to the user at the beginning and remove them when the classifier said that the answer is a good one.

To find out in which situation we are here, we have to understand how to measure precision and recall. And to understand that, we have to look into the four distinct classification results as they are described in the following table:

		Classified as	
		Positive	**Negative**
In reality it is	Positive	True positive (TP)	False negative (FN)
	Negative	False positive (FP)	True negative (TN)

For instance, if the classifier predicts an instance to be positive and the instance indeed is positive in reality, this is a true positive instance. If on the other hand the classifier misclassified that instance, saying that it is negative while in reality it was positive, that instance is said to be a false negative.

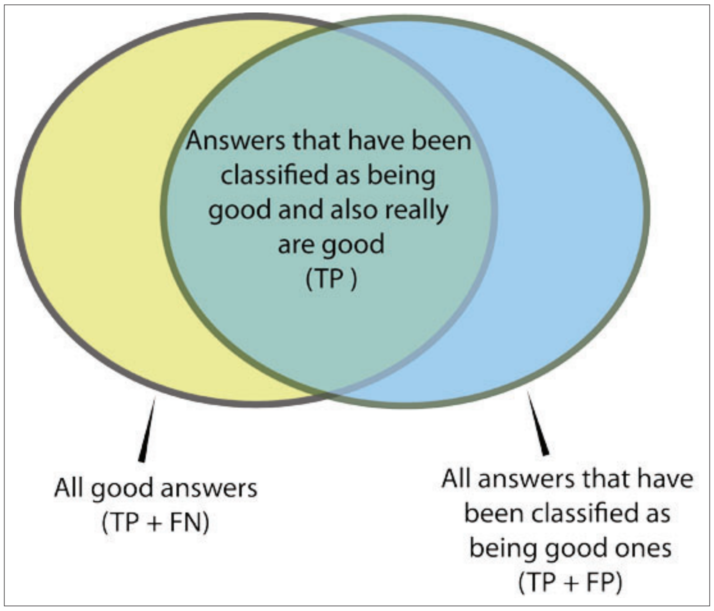
What we want is to have a high success rate when we are predicting a post as either good or bad, but not necessarily both. That is, we want as much true positives as possible. This is what precision captures:

$$Precision = \frac{TP}{TP + FP}$$

If instead our goal would have been to detect as much good or bad answers as possible, we would be more interested in recall:

$$Recall = \frac{TP}{TP + FN}$$

In terms of the following graphic, precision is the fraction of the intersection of the right circle while recall is the fraction of the intersection of the left circle:

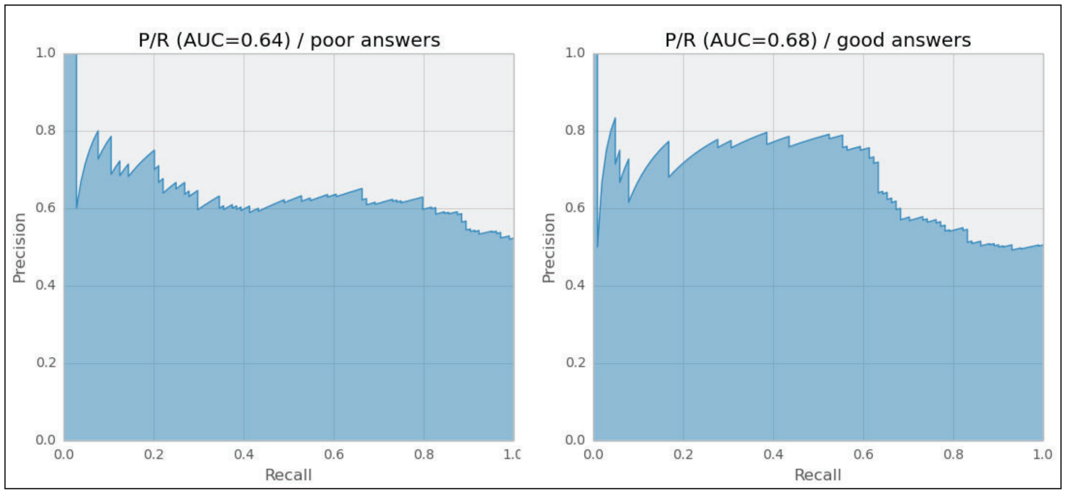

Answers that have been classified as being good and also really are good (TP)
All good answers (TP + FN)
All answers that have been classified as being good ones (TP + FP)

So, how can we now optimize for precision? Up to now, we have always used 0.5 as the threshold to decide whether an answer is good or not. What we can do now is count the number of TP, FP, and FN while varying that threshold between 0 and 1. With those counts, we can then plot precision over recall.

The handy function `precision_recall_curve()` from the metrics module does all the calculations for us:

```
>>> from sklearn.metrics import precision_recall_curve
>>> precision, recall, thresholds = precision_recall_curve(y_test,
    clf.predict(X_test))
```

Predicting one class with acceptable performance does not always mean that the classifier is also acceptable predicting the other class. This can be seen in the following two plots, where we plot the precision/recall curves for classifying bad (the left graph) and good (the right graph) answers:



In the graphs, we have also included a much better description of a classifier's performance, the **area under curve** (**AUC**). It can be understood as the average precision of the classifier and is a great way of comparing different classifiers.

We see that we can basically forget predicting bad answers (the left plot). Precision drops to a very low recall and stays at an unacceptably low 60 percent.

Predicting good answers, however, shows that we can get above 80 percent precision at a recall of almost 40 percent. Let's find out what threshold we need for that. As we trained many classifiers on different folds (remember, we iterated over `KFold()` a couple of pages back), we need to retrieve the classifier that was neither too bad nor too good in order to get a realistic view. Let's call it the medium clone:

```
>>> medium = np.argsort(scores)[int(len(scores) / 2)]
>>> thresholds = np.hstack(([0],thresholds[medium]))
>>> idx80 = precisions>=0.8
>>> print("P=%.2f R=%.2f thresh=%.2f" % (precision[idx80][0],
                                         recall[idx80][0], threshold[idx80]
[0]))
P=0.80 R=0.37 thresh=0.59
```

Setting the threshold at `0.59`, we see that we can still achieve a precision of 80 percent detecting good answers when we accept a low recall of 37 percent. That means that we would detect only one in three good answers as such. But from that third of good answers that we manage to detect, we would be reasonably sure that they are indeed good. For the rest, we could then politely display additional hints on how to improve answers in general.

To apply this threshold in the prediction process, we have to use `predict_proba()`, which returns per class probabilities, instead of `predict()`, which returns the class itself:

```
>>> thresh80 = threshold[idx80][0]
>>> probs_for_good = clf.predict_proba(answer_features)[:,1]
>>> answer_class = probs_for_good>thresh80
```

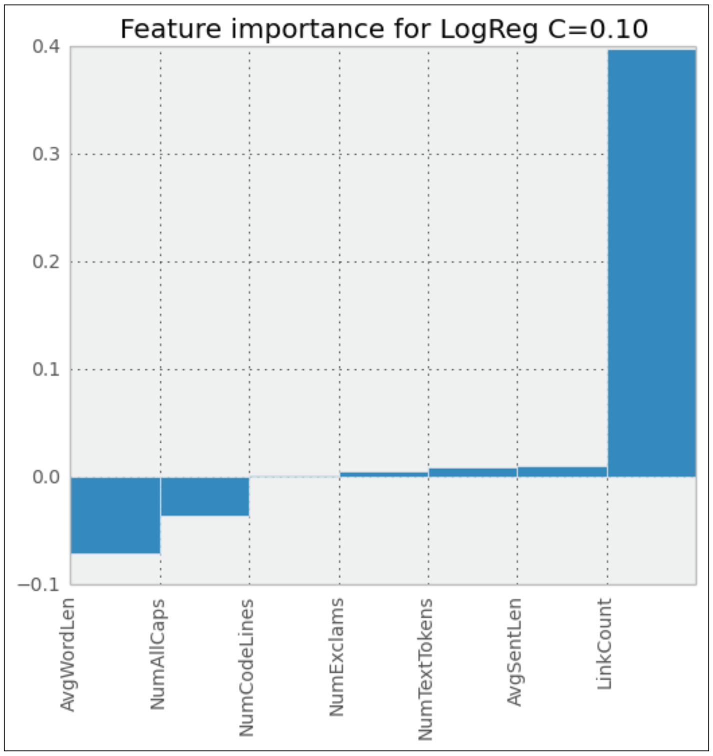
We can confirm that we are in the desired precision/recall range using `classification_report`:

```
>>> from sklearn.metrics import classification_report
>>> print(classification_report(y_test, clf.predict_proba [:,1]>0.63,
    target_names=['not accepted', 'accepted']))


            precision    recall  f1-score   support
not accepted         0.59      0.85      0.70       101
accepted             0.73      0.40      0.52        99
avg / total          0.66      0.63      0.61       200
```

Note that using the threshold will not guarantee that we are always above the precision and recall values that we determined above together with its threshold.

Slimming the classifier

It is always worth looking at the actual contributions of the individual features. For logistic regression, we can directly take the learned coefficients (`clf.coef_`) to get an impression of the features' impact. The higher the coefficient of a feature, the more the feature plays a role in determining whether the post is good or not. Consequently, negative coefficients tell us that the higher values for the corresponding features indicate a stronger signal for the post to be classified as bad.



We see that `LinkCount`, `AvgWordLen`, `NumAllCaps`, and `NumExclams` have the biggest impact on the overall classification decision, while `NumImages` (a feature that we sneaked in just for demonstration purposes a second ago) and `AvgSentLen` play a rather minor role. While the feature importance overall makes sense intuitively, it is surprising that `NumImages` is basically ignored. Normally, answers containing images are always rated high. In reality, however, answers very rarely have images. So, although in principal it is a very powerful feature, it is too sparse to be of any value. We could easily drop that feature and retain the same classification performance.

Ship it!

Let's assume we want to integrate this classifier into our site. What we definitely do not want is training the classifier each time we start the classification service. Instead, we can simply serialize the classifier after training and then deserialize on that site:

```
>>> import pickle
>>> pickle.dump(clf, open("logreg.dat", "w"))
>>> clf = pickle.load(open("logreg.dat", "r"))
```

Congratulations, the classifier is now ready to be used as if it had just been trained.

Summary

We made it! For a very noisy dataset, we built a classifier that suits a part of our goal. Of course, we had to be pragmatic and adapt our initial goal to what was achievable. But on the way we learned about strengths and weaknesses of nearest neighbor and logistic regression. We learned how to extract features such as `LinkCount`, `NumTextTokens`, `NumCodeLines`, `AvgSentLen`, `AvgWordLen`, `NumAllCaps`, `NumExclams`, and `NumImages`, and how to analyze their impact on the classifier's performance.

But what is even more valuable is that we learned an informed way of how to debug bad performing classifiers. That will help us in the future to come up with usable systems much faster.

After having looked into nearest neighbor and logistic regression, in the next chapter, we will get familiar with yet another simple yet powerful classification algorithm: Naïve Bayes. Along the way, we will also learn some more convenient tools from scikit-learn.