CHAPTER .1

Text Classification

Very often, we are interested in whether a document belongs to a particular class of interest: is it an ad or not (spam filtering)? Is it positive, negative, or neutral in tone (**sentiment analysis**)? Was it written by an older or a younger person (**author attribute prediction**)? Does it express any concepts we are interested in, say, whether it's a business-related text or not, the party it was most likely written by, or whether it addresses an audience or an individual? In order to get the answer to all of these questions, we use a statistical model (a **classifier**) that has been fitted on a training set by adjusting a set of parameters to maximize the number of correct matches.

sentiment analysis
author attribute prediction
classifier

Classification is one of the core machine learning applications and has gained a lot of attention over the last few year. It can seem complicated and mysterious, but all machine learning is closely related to common statistical methods used in social science. One of the most common (and useful) classification algorithms is in fact **logistic regression**, the same method used to compute the fit between a set of independent variables and a binary dependent variable in social sciences. The algorithm is exactly the same, but there are two main differences.

logistic regression

The first set of differences are terminological: instead of independent variables, machine learning talks about **features**; instead of dependent variables, it uses **targets**, whose values are called **classes**; and lastly, the coefficients (or betas) of the algorithm are referred to as **parameters**. Since logistic regression is used in both disciplines, we will use the respective terminology depending on the context.

features
targets
classes
parameters
weights, probabilities, and other settings

The second difference between the use of logistic regression as a classifier in machine learning and as a regression method in social science is its application: **prediction vs. explanation**. In social science, we fit the model to the data, and then examine the coefficients to find an explanatory causal correlation between the independent variables and the dependent variable. This model is rarely tested on new data. In machine learning, in contrast, the objective is only to perform well on new data. We freeze the coefficients after fitting, and measure how well the fitted model performs on new, held-out data. These two camps need not be an either-or: several researcher have shown (Shmueli et al., 2010; Hofman et al., 2017; Yarkoni and Westfall, 2017) that using machine learning methods and testing the predictive accuracy of a model is a good robustness test. Basically, both fields try to reverse-engineer the underlying data-generation process: ultimately, social science theory is also about predicting how systems (people, markets, firms) will behave under similar conditions in the future. The important difference is

prediction vs. explanation

that every causal model is also necessarily a good predictive model. The inverse, however, is not necessarily true (we will see this when we come to overfitting).

1. Terminology

There are a number of other classifiers common in machine learning (Naive Bayes, Support Vector Machines, Feed-Forward Neural Networks). The methods differ only in the ways they weigh and combine the input. However, the principle is always the same: find a pattern of weighted input features to map to the output target class. Therefore, all of them can be expressed as

$$y = f(X; \theta, b)$$

That is, the output value is a function $f(\cdot)$ of the input X, parametrized by a set of parameters, θ (depending on the algorithm, θ can be weights, probabilities, or activation functions), and a **bias term** b.

bias term,

In the case of a linear model, like logistic regression, $f(X)$ would be

$$\theta \cdot X + b$$

The output of this binary model is a value between 0 and 1. By checking against a threshold (typically 0.5), we can decide which of two classes to assign.

multi-class

If we have more than two classes (a **multi-class** problem, like in sentiment analysis, where we want to choose between three outcomes), we can simply train one binary model for each class, learning to distinguish instances of that class from all others (if we learn a multiclass problem in `sklearn`, this is what Python does under the hood). Alternatively, we can train a model that predicts a probability distribution over all the possible output values, and then choose the one with the highest value (called a **softmax**).

softmax

Formally, each training instance is a tuple

$$(X_i, y_i)$$

where X_i is a D-dimensional vector of indicators or real numbers describing the input, and $y \in K$ is the correct output class from a set of possible class labels, K. We also refer to the correct labels as **gold labels**. To distinguish the gold labels from the predictions of our model, we use a hat over the predictions: $\hat{y}$.

gold labels

We assume that the output is related to the input via some function that we need to find by fitting a number of parameters. This process is called **training**. The goal of training is to let the machine learn to find repeated patterns in the data that can be used to find the correct output for a new, held-out input example. This new sample is called the **test set**.

training

test set

What we are interested in here is to achieve the highest possible score of some appropriate **performance** metric on the held-out test data.

performance

2. Train-Dev-Test

We have already briefly discussed the role of the test data, the held-out sample that we measure performance on. But where does that data come from? After all, how can we measure performance on new data, if we don't know what the correct outputs for those new data points are? In order to simulate this condition, we divide our data set of (X_i, y_i) pairs into several parts. Before we start, we set aside some of the data that we will use as test, and do not use it until we are content with our model. We also can not include use the test set to collect all words in our vocabulary.

There is a problem with this setup, though. Models usually have a number of parameters we can set, and we will need to find the setting that works best for our particular data sample. If we fiddled long enough with the model parameters, and measured after each step how well the model does on the test data, we might get good performance on the test data, but we essentially cheated. We can only measure performance on test once!

To avoid this problem, we need another data set, which we can use to see how changing the model parameters affects performance on a held-out data set. This second held-out data set, which we use to tune our model parameters, is called the **development set** (or dev set for short).

development set

You can think about this as preparing for an important presentation to funders. You can only present to them once (i.e., they are your test set), but you might want to tweak a few things (those are your parameters). Since you cannot do a mock presentation with the funders, you ask some colleagues and friends. They are your development set.

Typically, you want your development and test set to be of equal size, have a representative distributions over outcomes, and be large enough to compute robust statistics. They need to be big enough to capture these conditions (if you only had 10 cases, your results might not be representative). It is hard to put a fixed number on this, but at the least, you should have several hundred instances for each of these data sets.

At the same time, you want your training data to be large enough to fit the model properly (if you had too few documents, you might never see enough combinations to set the parameters). Again, as a rough guideline, you should have thousands of instances. Common ratios of training:development:test data size are $80{:}10{:}10$, $70{:}15{:}15$ or $60{:}20{:}20$, depending on which split satisfies the above conditions best.

3. Cross-Validation

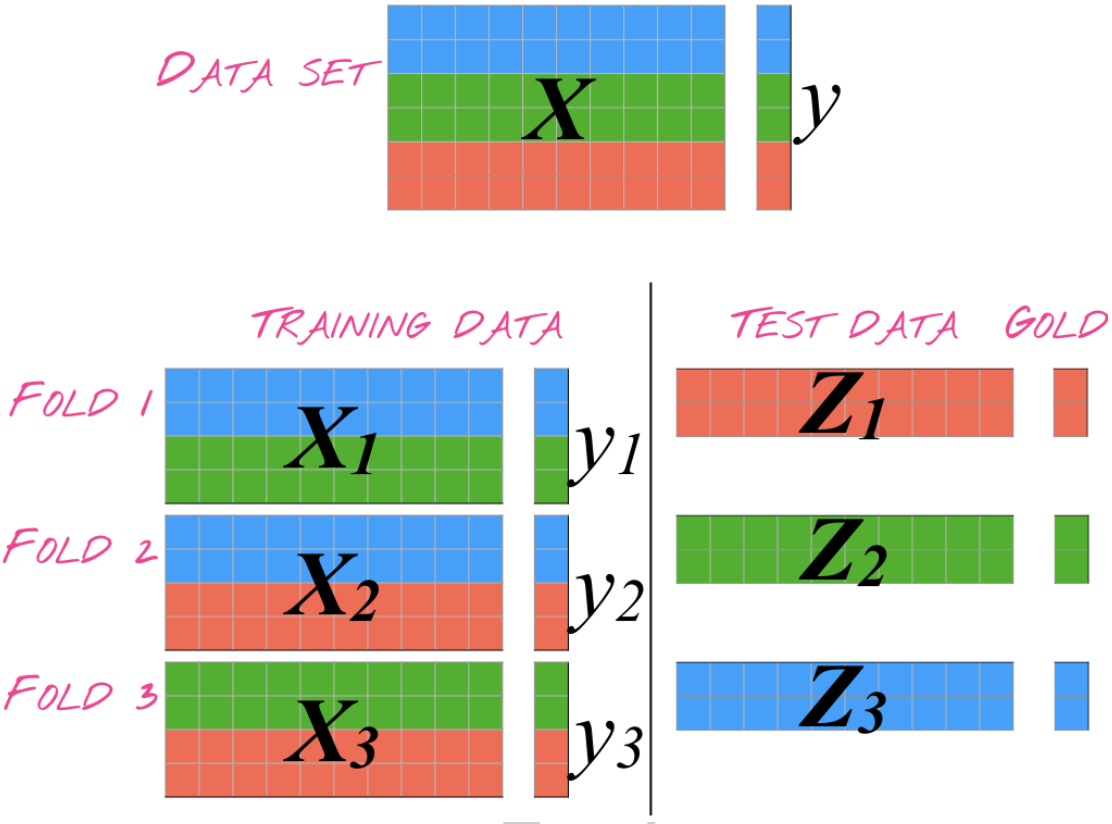

FIGURE 1. Schematic of 3-fold cross-validation

That is all fine and well, you might say, but I only have a data set of 1500 documents, and I cannot set aside some of them to use only once, or I reduce the amount of data used to fit the data. This is a good point.

k-fold cross-validation

For this case, you can use **k-fold cross-validation**. In k-fold cross-validation, we simulate new data by dividing the data into k parts, fitting k model on $k-1$ parts of the data, and evaluating on the k^th part. See Figure 1 for an example illustration with three folds. Each document has to end up in the held-out part once. In the end, we have performance scores from k models. The average over these scores tells us how well the model would work on new data.

This approach has several advantages: we can measure the performance on the entire data set, simulate different conditions, and therefore get a much more robust estimate. Again, choosing k depends on the resulting size of the training and test portions for each fold. k should be small enough to guarantee a decent-sized test set in each fold, but large enough to make the training part sufficient. Common values for k are 3, 5, or 10. In the extreme, if $k = N$, the size of our vocabulary, we fit one model and test it separately for each document in our corpus. This is also called **Leave-One-Out cross-validation**.

Leave-One-Out cross-validation*(LOO)*

If we want to have a dev set to tune each of the k models, we can further split the $k-1$ parts of the training portion in each fold. In that case, we train on $k-2$ parts, and tune and test on k parts each.

4. Evaluating Performance

In explanation, we are only interested in the fit of the model, typically the r^2 score. This is a performance metric, but it is difficult to generalize to held-out data.

Instead, in classification, we measure for how many of the documents in our test data a model produces the correct answer. Depending on how many classes we have, and balanced our data set is with respect to the distribution of these classes, there are different measures we can use. For all of them, we will use the three basic counts of **true positives** (TP: the model prediction and the true label are the same), **false positives** (the model predicts an instance to be of a certain class, but it is not), and **false negatives** (the model fails to recognize that an instance belongs to a certain class).

true positives
false positives
false negatives

The simplest measure of this performance is **accuracy**. We simply divide the number of true positives by the total number of instances, N.

accuracy

$$acc = \frac{TP + TN}{N}$$

Accuracy is a useful measure, but somewhat less useful when our labels are unbalanced: in a data set where 90% of all labels are class `A`, and 5% each are class `B` and `C`, a system that always predicts `A` would get an accuracy of 0.9 (all performance measures are usually given as floating point values between 0 and 1. Sometimes, people also report accuracy as percentage, though).

When our label distributino is more skewed, we need different measures than accuracy. **Precision** measures how many of our model's predictions were correct. We divide the number of true positives by the number of all positives.

Precision

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

In the case where we have only two classes that are equally frequent, precision is the same as accuracy.

Recall measures how many of the positive examples in the data our model managed to find. We divide the number of true positives by the sum of true positives (the instances our model got right) and false negatives (the instances our model *should* have gotten right, but did not).

Recall

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

A model that classified everything as, say, *positive*, would get a perfect recall for that class (it does, after all, find *all* true positives, while not producing any false negatives). However, such a model would obviously be useless, since its precision is bad.

F1 score

In order to get a single number, and to detect the case above, we want to balance precision and recall against each other. The **F1 score** does exactly that, by taking the harmonic mean between precision and recall.

$$F_1 = \frac{prec \times rec}{prec + rec}$$

Let's look at a number of examples (see Table 1). We will use the following data, from a hypothetical classifier that identifies animals (`1`) and natural objects (`0`).

X	y	$\hat{y}$	if target=1	if target=0
frog	1	1	TP	TN
deer	1	1	TP	TN
wolf	1	1	TP	TN
dog	1	1	TP	TN
bear	1	1	TP	TN
fish	1	1	TP	TN
bird	1	0	FN	FP
cat	1	0	FN	FP
stone	0	1	FP	FN
tree	0	0	TN	TP

TABLE 1. Example output of a classifier and their valuation for different target classes.

If we are interested in animals, our target label is `1`, and we get the following metrics from the measures in the 4th column:

$$acc = \frac{7}{10} = 0.7$$

$$prec = \frac{6}{7} = 0.86$$

$$rec = \frac{6}{8} = 0.75$$

$$F_1 = 0.81$$

If instead our target label is `0`, we get the following metrics:

$$acc = \frac{7}{10} = 0.7$$

$$prec = \frac{1}{3} = 0.33$$
$$rec = \frac{1}{2} = 0.5$$
$$F_1 = 0.4$$

Note that accuracy can always be used, and stays always the same, regardless of how many classes we have or what class we focus on: we only need to check whether the prediction and the gold data match. For precision, recall, and F1, we need to be calculate the score for each class separately. If we have *positive*, *negative*, and *neutral* as labels, and our target class is *positive*, either of the others is a false negative.

Even if we have several classes and compute precision, recall, and F1 separately, we might still want to have an overall metric. There are two ways of doing this: **micro-averaging** and **macro-averaging**.

micro-averaging

macro-averaging

In micro-averaging, we use the raw counts of the positives and negatives, and add them up. This gives us an average that is weighted by the class size, as larger classes have a greater influence on the outcome.

$$acc_{micro} = 7/10 + 7/10 = 14/20 = 0.7$$
$$prec_{micro} = 6/7 + 1/3 = 7/10 = 0.7$$
$$rec_{micro} = 6/8 + 1/2 = 7/10 = 0.7$$
$$F_{1micro} = 0.7$$

In macro-averaging, we first compute the individual scores for each class, and then take the average over them. This gives us an average that weighs all classes equally.

$$acc_{macro} = (0.7 + 0.7)/2 = 0.7$$
$$prec_{macro} = (0.86 + 0.33)/2 = 0.6$$
$$rec_{macro} = (0.5 + 0.75)/2 = 0.63$$
$$F_{1macro} = 0.61$$

Which averaging to use depends on the goal we have, and how much importance we attach to small classes.

5. Comparison and Significance Testing

When we train a classifier, we want to be able to use it in the future on unseen data. We have seen how we can measure and predict that performance, and how we can guard ourselves against overfitting.

However, while performance measures give us a good sense of the model's capabilities, it does not tell us anything about the difficulty of the classification task. If the task is extremely easy, a performance of 0.9 F1 score does not mean much.

To get a sense of the possible overhead, researchers often run a very simple **baseline** first. This can be an "algorithm" that always predicts the most frequent class label

baseline

majority baseline

(**majority baseline**), or another simple model. With non-linear models, we will see that it makes sense to compare against the baseline of a linear model. A model that does not manage to outperform the baseline is probably not on the right way, as it does not make sense to use a more complicated model where a simple one suffices.

Even if our model outperforms the baseline by a certain margin: does that mean another person running the model on their data is likely to see the same results? Given how sensitive the models are to domain differences (say, if we applied a model that was trained on newswire to Twitter data), there are not too many guarantees we can give. But say our user wants to apply the model to the same domain. Are the improvements over the baseline that we reported a fluke, or are they generalizable? This is essentially at the heart of **statistical significance** tests.

statistical significance

There are a great number of statistical significance tests out there, but it requires some knowledge to pick the right one for the task (Berg-Kirkpatrick et al., 2012). However, for many NLP problems, bootstrap sampling is a valid and interpretable test. The intuition is simple: when we compare the results of our model against a baseline, we only have one sample, which might be biased. So the result could very well be due to the particular composition of the data, and disappear on a different data set. Short of testing on a second sample, we can simulate different data sets by repeatedly sampling from the data with replacement (which is called **bootstrapping**), and comparing the baseline and model on each of those samples. We can now measure how many times the difference between the model and baseline is more extreme. The **central limit theorem** describes how the results of repeated measures from a population follow a normal distribution (think of a stadium full of people, from which you repeatedly draw 5 and measure their average height: most of the time, you will get an average height, and only rarely will you get all the tall people). We can use this insight to measure difference from the base case in two ways: the number of samples deviating more than two standard deviations from the original difference between the systems, or negative differences (meaning the better system differs from the base case). The latter case might seem more intuitive, but requires some assumptions (the sample mean is the same as the difference on the entire data set, and the distribution over differences is symmetrical), under which it is actually equivalent to the version here.

bootstrapping

central limit theorem

Take the examples in Table 2: on the entire sample, system A is 0.24 points better than system B. If we take enough samples, the average difference over those samples will be very similar. The question we are asking with bootstrap sampling is: *in how many of these samples is the difference at least twice as extreme as on the base case*? This setup requires us to put the system that has a higher score on the entire data set first. We collect 10 samples here (this is only for demonstration purposes, normally we would collect at least 10,000).

Figure 2 shows the distribution over the observed differences. As expected, their average difference is about 0.24, the same as the difference between the systems on the entire data. However, in three cases (sample 5, 8, and 10), the difference is more than

	A	B	difference
base	82.13	81.89	0.24
1	81.96	82.03	-0.07
2	81.86	82.61	-0.75
3	81.70	81.44	0.26
4	82.42	82.77	-0.35
5	81.89	81.06	0.83
6	81.39	81.24	0.15
7	81.96	81.58	0.37
8	82.57	81.65	0.92
9	82.50	82.67	-0.17
10	83.07	81.84	1.23

TABLE 2. Example performance of two systems on the entire data set (base) and on 10 samples.

twice the base difference. Because the sampled differences follow a normal distribution with a small mean, there are also some cases where A is actually worse than B, so this indicates that the observed difference is not significant. The p-value here would be $3/10 = 0.3$. As the base difference increases, there are fewer and fewer extreme cases.

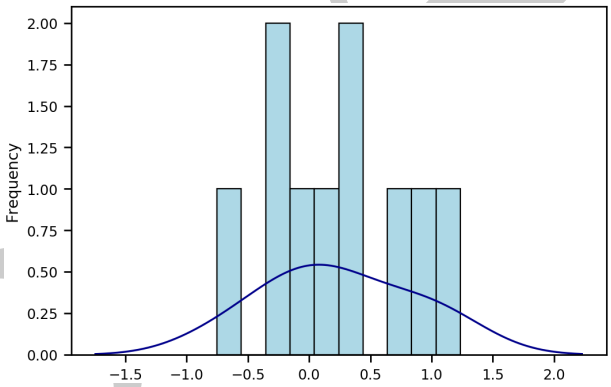

FIGURE 2. Distribution over differences between the two systems on 10 samples

```python
import numpy as np

def bootstrap_sample(system1, system2, gold, samples=10000, score=
    precision_score):
    """
    compute the proportion of times the performance difference of the
    two systems on a subsample is significantly different from the
```

```
    performance on the entire sample
    """
    N = len(gold) # number of instances

    # make sure the two systems have the same number of samples
    assert len(system1) == N and len(system2) == N, 'samples have different lengths'

    # compute performance score on entire sample
    base_score1 = score(gold, system1, average='binary')
    base_score2 = score(gold, system2, average='binary')

    # compute the difference
    basedelta = base_score1 - base_score2
    assert basedelta > 0, 'Wrong system first, system1 needs to be better!'

    system1 = np.array(system1)
    system2 = np.array(system2)
    gold = np.array(gold)

    p = 0
    for i in range(samples):
        # select a subsample, with replacement
        sample = np.random.choice(N, size=N, replace=True)

    # collect data corresponding to subsample
        sample1 = system1[sample]
        sample2 = system2[sample]
        gold_sample = gold[sample]

    # compute scores on subsample
        sample_score1 = score(gold_sample, sample1, average='binary')
        sample_score2 = score(gold_sample, sample2, average='binary')
        sample_delta = sample_score1 - sample_score2

        # check whether the observed sample difference is at least
        # twice as large as the base difference
        if sample_delta > 2*basedelta:
            p += 1

    return p/samples
```

CODE 1. Bootstrap sampling for statistical significance tests.

6. Overfitting and Regularization

Imagine a (slightly contrived) case where for some reason, most positive training examples in a text classification task contained the word "Tuesday". From the perspective of the classifier, the presence of "Tuesday" is therefore an excellent feature to predict positive instances. But if the new instances we encounter in our test set never contain the word "Tuesday", the model comes up empty-handed when it tries to classify them. It has been **overfitting** to the training data. This is similar to memorizing vs. understanding: if we memorize that $2 + 2 = 4$, we might be surprised when we are told that $3 + 1$ is also 4. If instead we understand how addition works, we will be much better equipped to handle future cases.

overfitting

Training (i.e., fitting) a model on too many features can cause the model to memorize parts of the training data. While that is excellent for predicting *those* outcomes correctly, it is entirely useless when encountering new examples.

In most social science scenarios, we only use a small number of independent variables, but in machine learning, we often use thousands of them. When working with text, one of the easiest ways to **featurize** the input is to make each observed word type a feature. Depending on the estimate, English has between $100,000$ and $500,000$ words, so the number of features quickly becomes huge. This number of independent variables is frowned upon in social sciences, and for good reason.

featurize

In the extreme case, if we have more features than observations, we can simply find one feature that explains each observation. As a result, it is hard to find any one independent variable In machine learning, this situation is also frowned upon, but for different reasons.

In prediction, we would like to prevent the model from perfectly memorizing the training data. Earlier, we mentioned the bias term, b in our models. This is where we come back to it. Choosing a bias that prevents the model from overfitting is called **regularization**. It is always a good idea to regularize the models, even if we do not want to use them for prediction.

regularization

We can think of regularization as a way to make training harder, so that the model is more prepared to handle different scenarios in the future. It is a bit like going running with a snorkel, or swimming with a buoy in tow: by learning to overcome these conditions in training, we are better equipped for the real thing.

The simplest way to use a bias term is to add some random noise to the data. Because the error is random, the model can never fit it perfectly, and so we help avoid overfitting. In social science, the bias term is therefore often called an "**error term**", since we assume that there is some measurement associated with our inputs.

error term

However, randomness is hard for computers, so we often use normally (in the sense of Gaussian) distributed noise, which in itself is regular. Furthermore, it is a strong assumption to think the noise we encounter is normally distributed: In the case of BOW representations, there should not be any noise in the first place - we simply use word

counts. In other cases, the error might actually follow a power-law distribution.

Instead of guessing at the error distribution, we can tie the bias term to the distribution of parameters. Ideally, we would like a model that considers all features, rather than putting all its eggs in one basket (as in the "Tuesday" example above).

Essentially, such a model would distribute the weights for each feature relatively evenly.

L2 norm

We can measure this by looking at the **L2 norm** of the weight vector: the root of the sum of squares of all weights. The more evenly distributed the weights are, the smaller this term becomes. The effect is that the model is forced to "hedge its bets" by considering *all* features, rather than putting all its faith in a few of them, and is therefore better equipped to deal with future cases. This process is called **L2 regularization**. The regularization term is usually weighted by a constant called λ.

L2 regularization

$$y = W \cdot \mathbf{X} + \lambda ||W||_2$$

The L2 norm uses the root of the sum of squares as error term, to remove any signs on the elements:

$$||W||_2 = \sqrt{\sum_{i=1}^{N} w_i^2}$$

The L2 norm has has unique analytical solutions, but it produces non-sparse coefficients (every element has a non-zero value), and therefore does not perform variable selection.

Ridge regression

When we use L2 as regularization for Logistic Regression, it is also known as **Ridge regression**.

L1 regularization

Theoretically, we can also minimize the L1 norm of the vector (**L1 regularization**). Here, we use the regular norm of the coefficient vector as error term:

$$||W||_1 = \sum_{i=1}^{N} |w_i|$$

The L1 norm produces sparse coefficients, i.e., it sets many coefficients to 0, and therefore has many possible solutions for a given vector. Due to the sparsity of the vectore, they all amount to an implicit variable selection: we can ignore all entries that have a value of 0, and focus on the others. When using L1 regularization for Logistic regression, this is also known as **Lasso regression**.

Lasso regression

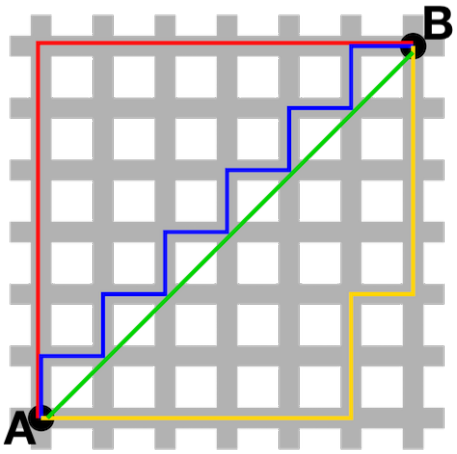

FIGURE 3. Examples of L1 vs. L2 norm in Euclidean space (modified from Wikipedia)

To see the difference between the two norms, see Figure 3: the red, blue, and yellow lines are all possible L1 solutions with the same value (12), whereas the green line is the L2 solution (8.49). Norms are closely related to distance measures like the Manhattan distance (which is equivalent to the L1 norm) or the Euclidean distance (which is equivalent to the L2 norm).

More rarely, we might use the **L0 norm** for regularization, which is the sum of all elements in the weight vector raised to their 0th power (i.e., either 0 or 1), or simply the number of non-zero coefficients. It's a very aggressive way of eliminating uninformative features. **L0 norm**

In practice, L2 regularization works best for prediction, though. In `sklearn`, logistic regression is already set to use L2 regularization, with a `C` parameter that stands for the λ weight. L1 is most useful in feature selection (see page 14).

7. Feature Selection

We have our data, we have decided on a classifier, and made sure it does not overfit. However, we are not sure the performance is maxed out with these choices. Can we do more to improve performance?

If we are using a BOW representation and a linear model, we can fit a classifier (with regularization!), and then analyze then inspect the magnitude of the coefficients. This **feature selection** can assist us in improving performance, by choosing a smaller, more informative set of features. Naturally, if we use feature selection to improve performance, we have to do it on the training data, and cannot peak at the development or test data!

feature selection

However, analyzing features can give us an exploratory overview of the correlation between our features and the target class. This can be informative to find out, for example, which words best describe each political candidate in a primary (see `https://www.washingtonpost.com/news/monkey-cage/wp/2016/02/24/these-6-charts-show-how-much-sexism-hillary-clinton-faces-on-twitter`).

7.1. Dimensionality Reduction. We have looked at dimensionality reduction earlier in the context of visualization (see page **??**). However, we can also use it to improve our performance. Rather than fitting a model on a large set of features, we can use dimensionality reduction techniques to find a smaller number of dimensions that still capture the variation in the data, but provide fewer chances to overfit by exploiting spurious patterns in the data. This could for example be the $\mathbf{U}$ matrix from SVD.

If we use a lower-dimensional version of a BOW matrix (i.e., counts or TFIDF values), the individual dimensions (the columns of our matrix) do not mean anything anymore: we cannot map them back to the terms we used as features.

7.2. Chi-Squared. ξ^2 (or Chi squared) is a test that measures the correlation between a positive feature and the categorical outcome variable.

In `sklearn`, we can use the `feature_selection` package to implement the selection process with ξ^2, by sorting the features according to their correlation, and keeping only the top k.

```
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2

selector = SelectKBest(chi2, k=1500).fit(X, y)
X_sel = selector.transform(X)
print(X_sel.shape)
```

CODE 2. Selecting the top 1500 features according to their ξ^2 value.

7.3. Randomized Regression. Logistic regression is a generalized linear model, where the log odds of the response probability is a linear function of the independent variables. The coefficients of these variables are estimated via maximum likelihood, and are then subsequently interpreted. If we use the model on word count data and a target category, we can analyze the importance of individual terms by inspecting the magnitude of the coefficients for those terms.

However, this approach presupposes that we have already selected the independent variables. In the case of text, however, where the vocabulary size, and therefore the number of independent variables, can be massive, yet their individual occurrences sparse (some words occur only once even in large corpora), it is often impossible to know a priori which variables to select.

There are several ways in which we can potentially select a subset of informative variables: We can remove variables with non-significant co-occurrence statistics. However, this requires a large data set and does not guarantee the best model. We can fit all possible models on the entire data and pick the one with the best fit, using some measure of fitness. However, due to the large number of possible models, this is computationally prohibitively extensive and not robust to overfitting. We can fit a model on the entire data, and use an L1 penalty term. This constraint causes uninformative variables to receive 0 weight during fitting. However, this method is highly reliant on large data and sufficient observations. To address the last concern, we can randomize the previous method. We collect repeated random subsets of the data, fit a L1-penalized (Lasso) model to each and aggregate the coefficient vectors. Variables that frequently receive a high coefficient are then selected. When the target variable is binary, this process is randomized logistic regression (or Stability Selection, see Meinshausen and Bühlmann (2010))

Intuitively, we simulate a range of different conditions, and pick the variables that are informative independent of the condition. In practice, we fit 200-1000 independent Logistic Regression models on the data, each on a random subset sampled with replacement. For each model, we use a different weight for the L1 regularization parameter, controlling how aggressively coefficients are driven to 0. Variables that have a score of at least 0.5 (i.e., they received a positive coefficient weight in over 50% of the models) can be considered informative.

```
1 from sklearn.linear_model import LogisticRegression
2 import numpy as np
3
4 def positive_indicatorsRLR(X, y, target, vectorizer,
     selection_threshold=0.3, num_iters=100):
5     n_instances, n_feats = X.shape
6
7     pos_scores = [] # all coefficient > 0
8     neg_scores = [] # all coefficient < 0
9     # choices for lambda weight
```

```
    penalities = [10,5,2,1,0.5,0.1,0.05,0.01,0.005,0.001,0.0001,
0.00001]

    # select repeated subsamples
    for iteration in range(num_iters):
        # initialize a model with randomly-weighted L1 penalty
        clf = LogisticRegression(penalty='l1', C=penalities[np.random
.randint(len(penalities))])

    # choose a random subset of indices of the data with replacement
        selection = np.random.choice(n_instances, size=int(
n_instances * 0.75))
        try:
            clf.fit(X[selection], y[selection])
        except ValueError:
            continue

    # record which coefficients got a positive or negative score
        pos_scores.append(clf.coef_ > 0)
        neg_scores.append(clf.coef_ < 0)

    # normalize the counts
    pos_scores = (np.array(pos_scores).sum(axis=0)/num_iters).reshape
(-1)
    neg_scores = (np.array(neg_scores).sum(axis=0)/num_iters).reshape
(-1)

    # find the features corresponding to the non-zero coefficients
    features = vectorizer.get_feature_names()
    pos_positions = [i for i, v in enumerate(pos_scores >=
selection_threshold) if v]
    neg_positions = [i for i, v in enumerate(neg_scores >=
selection_threshold) if v]

    pos = [(features[i], pos_scores[i]) for i in pos_positions]
    neg = [(features[i], neg_scores[i]) for i in neg_positions]

    posdf = pd.DataFrame(pos, columns='term score'.split()).
sort_values('score', ascending=False)
    negdf = pd.DataFrame(neg, columns='term score'.split()).
sort_values('score', ascending=False)

    return posdf, negdf
```

CODE 3. Randomized Regression for variable selection.

8. Retrofitting to Increase Within-Class Similarity

We have previously seen (page **??**) how we can use retrofitting to include non-linguistic information in embeddings. However, the same technique can be used to make classes more linearly separable Hovy and Purschke (2018). Figure 4 shows an example for 500 data points from ten classes (each class is labeled with a different color). Retrofitting sorts the instances into clusters based on color.

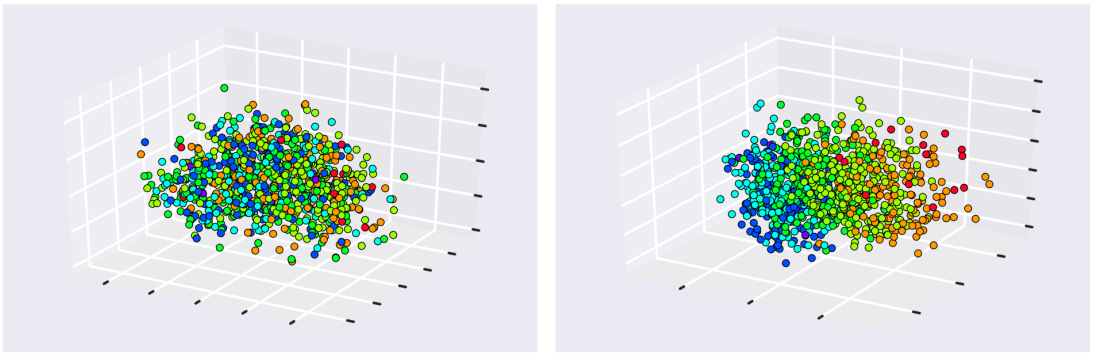

FIGURE 4. Effect of retrofitting on 500 instances in a ten-class embedding space. Each color represents one class label. Before retrofitting (left), instances are based on linguistic information. After retrofitting on labels (right), classes are more clearly separated.

Instead of external information, we can also use the class labels as equivalence classes, i.e., $\Omega = \{(d_i, d_j | y_i = y_j)\}$. By defining the neighborhood of each instance to be all instances with the same class label, we basically separate out classes in the embedding space, making them easier to differentiate by a classifier. This effectively improves classification performance.

However, we can only retrofit the embeddings of authors in the training set D_{train}, since we need information about the class label in order to construct Ω. However, the retrofitting process changes the configuration of the embedding space (into $\hat{D_{train}}$), so a separating hyperplane learned on $\hat{D}_{train}$ will not be applicable to a test set D_{test} in the original embedding space.

In order to extend the homophily information to instances in the test set, we use a *translation matrix* T (a 300×300 matrix), which approximates the transformation from the original training data matrix D_{train} into the retrofitted matrix $\hat{D}_{train}$. We obtain T by minimizing the least-square difference in $D_{train} \cdot T = \hat{D}_{train}$.

T captures the retrofitting operation, and allows us to modify the test instances' representations *as if* their classes were known, despite the absence of label information. In particular, by applying T to the embeddings in the unlabeled test set D_{test}, we obtain a retrofitted version $\hat{D_{test}}$ that preserves the transformation learned on the training data. Since the least-square approximation is not perfect, we find that in practice fitting a

classifier on the approximation $D_{train} \cdot T$ works better than using $\hat{D}_{train}$, acting as a regularizer.

Bibliography

Taylor Berg-Kirkpatrick, David Burkett, and Dan Klein. 2012. An empirical investigation of statistical significance in NLP. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 995–1005. Association for Computational Linguistics.

Jake M Hofman, Amit Sharma, and Duncan J Watts. 2017. Prediction and explanation in social systems. *Science*, 355(6324):486–488.

Dirk Hovy and Christoph Purschke. 2018. Capturing regional variation with distributed place representations and geographic retrofitting. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 4383–4394.

Nicolai Meinshausen and Peter Bühlmann. 2010. Stability selection. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(4):417–473.

Galit Shmueli et al. 2010. To explain or to predict? *Statistical science*, 25(3):289–310.

Tal Yarkoni and Jacob Westfall. 2017. Choosing prediction over explanation in psychology: Lessons from machine learning. *Perspectives on Psychological Science*, 12(6):1100–1122.