How to make a racist AI without really trying

Robyn Speer — July 13, 2017

A cautionary tutorial.

Let's make a sentiment classifier!

Sentiment analysis is a very frequently-implemented task in NLP, and it's no surprise. Recognizing whether people are expressing positive or negative opinions about things has obvious business applications. It's used in social media monitoring, customer feedback, and even automatic stock trading (leading to bots that buy Berkshire Hathaway when Anne Hathaway gets a good movie review).

It's simplistic, sometimes too simplistic, but it's one of the easiest ways to get measurable results from NLP. In a few steps, you can put text in one end and get positive and negative scores out the other, and you never have to figure out what you should do with a parse tree or a graph of entities or any difficult representation like that.

So that's what we're going to do here, following the path of least resistance at every step, obtaining a classifier that should look very

familiar to anyone involved in current NLP. For example, you can find this model described in the Deep Averaging Networks paper (Iyyer et al., 2015). This model is not the point of that paper, so don't take this as an attack on their results; it was there as an example of a well-known way to use word vectors.

Here's the outline of what we're going to do:

- Acquire some typical **word embeddings** to represent the meanings of words
- Acquire **training and test data**, with gold-standard examples of positive and negative words
- **Train a classifier**, using gradient descent, to recognize other positive and negative words based on their word embeddings
- Compute **sentiment scores** for sentences of text using this classifier
- **Behold the monstrosity** that we have created

And at that point we will have shown "how to make a racist AI without really trying". Of course that would be a terrible place to leave it, so afterward, we're going to:

- **Measure the problem** statistically, so we can recognize if we're solving it
- **Improve the data** to obtain a semantic model that's more accurate *and* less racist

Software dependencies

This tutorial is written in Python, and relies on a typical Python

machine-learning stack: `numpy` and `scipy` for numerical computing, `pandas` for managing our data, and `scikit-learn` for machine learning. Later on we'll graph some things with `matplotlib` and `seaborn`.

You could also replace `scikit-learn` with TensorFlow or Keras or something like that, as they can also train classifiers using gradient descent. But there's no need for the deep-learning abstractions they provide, as it only takes a single layer of machine learning to solve this problem.

```
import numpy as np
import pandas as pd
import matplotlib
import seaborn
import re
import statsmodels.formula.api

from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

```
# Configure how graphs will show up in this notebook
%matplotlib inline
seaborn.set_context('notebook', rc={'figure.figsize': (10, 6)}, font_
scale=1.5)
```

Step 1: Word embeddings

Word embeddings are frequently used to represent words as inputs to machine learning. The words become vectors in a multi-dimensional space, where nearby vectors represent similar meanings. With word embeddings, you can compare words by (roughly) what they mean, not just exact string matches.

Successfully training word vectors requires starting from hundreds

of gigabytes of input text. Fortunately, various machine-learning groups have already done this and provided pre-trained word embeddings that we can download.

Two very well-known datasets of pre-trained English word embeddings are **word2vec**, pretrained on Google News data, and **GloVe**, pretrained on the Common Crawl of web pages. We would get similar results for either one, but here we'll use GloVe because its source of data is more transparent.

GloVe comes in three sizes: 6B, 42B, and 840B. The 840B size is powerful, but requires significant post-processing to use it in a way that's an improvement over 42B. The 42B version is pretty good and is also neatly trimmed to a vocabulary of 1 million words. Because we're following the path of least resistance, we'll just use the 42B version.

> **Why does it matter that the word embeddings are "well-known"?**
>
> I'm glad you asked, hypothetical questioner! We're trying to do something extremely typical at each step, and for some reason, comparison-shopping for better word embeddings isn't typical yet. Read on, and I hope you'll come out of this tutorial with the desire to use modern, high-quality word embeddings, especially those that are aware of algorithmic bias and try to mitigate it. But that's getting ahead of things.

We download glove.42B.300d.zip from the GloVe web page, and extract it into `data/glove.42B.300d.txt`. Next we define a function to read the simple format of its word vectors.

```
def load_embeddings(filename):
```

```
    """
    Load a DataFrame from the generalized text format used by word2ve
c, GloVe,
    fastText, and ConceptNet Numberbatch. The main point where they d
iffer is
    whether there is an initial line with the dimensions of the matri
x.
    """
    labels = []
    rows = []
    with open(filename, encoding='utf-8') as infile:
        for i, line in enumerate(infile):
            items = line.rstrip().split(' ')
            if len(items) == 2:
                # This is a header row giving the shape of the matrix
                continue
            labels.append(items[0])
            values = np.array([float(x) for x in items[1:]], 'f')
            rows.append(values)

    arr = np.vstack(rows)
    return pd.DataFrame(arr, index=labels, dtype='f')

embeddings = load_embeddings('data/glove.42B.300d.txt')
embeddings.shape
```

```
(1917494, 300)
```

Step 2: A gold-standard sentiment lexicon

We need some input about which words are positive and which words are negative. There are many sentiment lexicons you could use, but we're going to go with a very straightforward lexicon (Hu and Liu, 2004), the same one used by the Deep Averaging Networks paper.

We download the lexicon from Bing Liu's web site (https://www.cs.uic.edu/~liub/FBS/sentiment-analysis.html#lexicon) and extract it into `data/positive-words.txt` and `data/negative-words.txt`.

Next we define how to read these files, and read them in as the `pos_words` and `neg_words` variables:

```
def load_lexicon(filename):
    """
    Load a file from Bing Liu's sentiment lexicon
    (https://www.cs.uic.edu/~liub/FBS/sentiment-analysis.html), containing
    English words in Latin-1 encoding.

    One file contains a list of positive words, and the other contains
    a list of negative words. The files contain comment lines starting
    with ';' and blank lines, which should be skipped.
    """
    lexicon = []
    with open(filename, encoding='latin-1') as infile:
        for line in infile:
            line = line.rstrip()
            if line and not line.startswith(';'):
                lexicon.append(line)
    return lexicon

pos_words = load_lexicon('data/positive-words.txt')
neg_words = load_lexicon('data/negative-words.txt')
```

Step 3: Train a model to predict word sentiments

Our data points here are the embeddings of these positive and negative words. We use the Pandas `.loc[]` operation to look up the embeddings of all the words.

Some of these words are not in the GloVe vocabulary, particularly the misspellings such as “fancinating”. Those words end up with rows full of `NaN` to indicate their missing embeddings, so we use `.dropna()` to remove them.

```
pos_vectors = embeddings.loc[pos_words].dropna()
neg_vectors = embeddings.loc[neg_words].dropna()
```

Now we make arrays of the desired inputs and outputs. The inputs are the embeddings, and the outputs are 1 for positive words and -1 for negative words. We also make sure to keep track of the words they're labeled with, so we can interpret the results.

```
vectors = pd.concat([pos_vectors, neg_vectors])
targets = np.array([1 for entry in pos_vectors.index] + [-1 for entry
 in neg_vectors.index])
labels = list(pos_vectors.index) + list(neg_vectors.index)
```

> **Hold on. Some words are neither positive nor negative, they're neutral. Shouldn't there be a third class for neutral words?**
>
> I think that having examples of neutral words would be quite beneficial, especially because the problems we're going to see come from assigning sentiment to words that shouldn't have sentiment. If we could reliably identify when words should be neutral, it would be worth the slight extra complexity of a 3-class classifier. It requires finding a source of examples of neutral words, because Liu's data only lists positive and negative words.
>
> So I tried a version of this notebook where I put in 800 examples of neutral words, and put a strong weight on predicting words to be neutral. But the end results were not much different from what you're about to see.
>
> **How is this list drawing the line between positive and negative anyway? Doesn't that depend on context?**
>
> Good question. Domain-general sentiment analysis isn't as straightforward as it sounds. The decision boundary we're trying to find is fairly arbitrary in places. In this list, "audacious" is marked as "bad" while "ambitious" is "good". "Comical" is bad, "humorous" is good. "Refund" is good, even though it's typically in bad situations that you

> have to request one or pay one.
>
> I think everyone knows that sentiment requires context, but when implementing an easy approach to sentiment analysis, you just have to kind of hope that you can ignore context and the sentiments will average out to the right trend.

Using the scikit-learn `train_test_split` function, we simultaneously separate the input vectors, output values, and labels into training and test data, with 10% of the data used for testing.

```
train_vectors, test_vectors, train_targets, test_targets, train_label
s, test_labels = \
    train_test_split(vectors, targets, labels, test_size=0.1, random_
state=0)
```

Now we make our classifier, and train it by running the training vectors through it for 100 iterations. We use a logistic function as the loss, so that the resulting classifier can output the probability that a word is positive or negative.

```
model = SGDClassifier(loss='log', random_state=0, n_iter=100)
model.fit(train_vectors, train_targets)
```

```
SGDClassifier(alpha=0.0001, average=False, class_weight=None, epsilon=
0.1,
       eta0=0.0, fit_intercept=True, l1_ratio=0.15,
       learning_rate='optimal', loss='log', n_iter=100, n_jobs=1,
       penalty='l2', power_t=0.5, random_state=0, shuffle=True, verbos
e=0,
       warm_start=False)
```

We evaluate the classifier on the test vectors. It predicts the correct sentiment for sentiment words outside of its training data 95% of the time. Not bad.

```
accuracy_score(model.predict(test_vectors), test_targets)
```

```
0.95022624434389136
```

Let's define a function that we can use to see the sentiment that this classifier predicts for particular words, then use it to see some examples of its predictions on the test data.

```
def vecs_to_sentiment(vecs):
    # predict_log_proba gives the log probability for each class
    predictions = model.predict_log_proba(vecs)

    # To see an overall positive vs. negative classification in one n
umber,
    # we take the log probability of positive sentiment minus the log
    # probability of negative sentiment.
    return predictions[:, 1] - predictions[:, 0]


def words_to_sentiment(words):
    vecs = embeddings.loc[words].dropna()
    log_odds = vecs_to_sentiment(vecs)
    return pd.DataFrame({'sentiment': log_odds}, index=vecs.index)


# Show 20 examples from the test set
words_to_sentiment(test_labels).ix[:20]
```

	sentiment
fidget	-9.931679
interrupt	-9.634706
staunchly	1.466919
imaginary	-2.989215
taxing	0.468522
world-famous	6.908561
low-cost	9.237223

disapointment	-8.737182
totalitarian	-10.851580
bellicose	-8.328674
freezes	-8.456981
sin	-7.839670
fragile	-4.018289
fooled	-4.309344
undecided	-2.816172
handily	2.339609
demonizes	-2.102152
easygoing	8.747150
unpopular	-7.887475
commiserate	1.790899

More than the accuracy number, this convinces us that the classifier is working. We can see that the classifier has learned to generalize sentiment to words outside of its training data.

Step 4: Get a sentiment score for text

There are many ways to combine sentiments for word vectors into an overall sentiment score. Again, because we're following the path of least resistance, we're just going to average them.

```
import re
TOKEN_RE = re.compile(r"\w.*?\b")
# The regex above finds tokens that start with a word-like character
(\w), and continues
# matching characters (.+?) until the next word break (\b). It's a re
```

```
latively simple
# expression that manages to extract something very much like words f
rom text.


def text_to_sentiment(text):
    tokens = [token.casefold() for token in TOKEN_RE.findall(text)]
    sentiments = words_to_sentiment(tokens)
    return sentiments['sentiment'].mean()
```

There are many things we could have done better:

- Weight words by their inverse frequency, so that words like "the" and "I" don't cause big changes in sentiment
- Adjust the averaging so that short sentences don't end up with the most extreme sentiment values
- Take phrases into account
- Use a more robust word-segmentation algorithm that isn't confused by apostrophes
- Account for negations such as "not happy"

But all of those would require extra code and wouldn't fundamentally change the results we're about to see. At least now we can roughly compare the relative positivity of different sentences:

```
text_to_sentiment("this example is pretty cool")
```

```
3.889968926086298
```

```
text_to_sentiment("this example is okay")
```

```
2.7997773492425186
```

```
text_to_sentiment("meh, this example sucks")
```

```
-1.1774475917460698
```

Step 5: Behold the monstrosity that we have created

Not every sentence is going to contain obvious sentiment words.

Let's see what it does with a few variations on a neutral sentence:

```
text_to_sentiment("Let's go get Italian food")
```

```
2.0429166109408983
```

```
text_to_sentiment("Let's go get Chinese food")
```

```
1.4094033658140972
```

```
text_to_sentiment("Let's go get Mexican food")
```

```
0.38801985560121732
```

This is analogous to what I saw when I experimented with analyzing restaurant reviews using word embeddings, and found out that all the Mexican restaurants were ending up with lower sentiment for no good reason.

Word vectors are capable of representing subtle distinctions of meaning just by reading words in context. So they're also capable of representing less-subtle things like the biases of our society.

Here are some other neutral statements:

```
text_to_sentiment("My name is Emily")
```

```
2.2286179364745311
```

```
text_to_sentiment("My name is Heather")
```

```
1.3976291151079159
```

```
text_to_sentiment("My name is Yvette")
```

```
0.98463802132985556
```

```
text_to_sentiment("My name is Shaniqua")
```

```
-0.47048131775890656
```

Well, dang.

The system has associated wildly different sentiments with people's names. You can look at these examples and many others and see that the sentiment is generally more positive for stereotypically-white names, and more negative for stereotypically-black names.

This is the test that Caliskan, Bryson, and Narayanan used to conclude that semantics derived automatically from language corpora contain human-like biases, a paper published in *Science* in April 2017, and we'll be using more of it shortly.

Step 6: Measure the problem

We want to learn how to not make something like this again. So let's put more data through it, and statistically measure how bad its bias is.

Here we have four lists of names that tend to reflect different ethnic backgrounds, mostly from a United States perspective. The first two are lists of predominantly "white" and "black" names adapted from Caliskan et al.'s article. I also added typically Hispanic names, as well as Muslim names that come from Arabic or Urdu; these are two more distinct groupings of given names that tend to represent your background.

This data is currently used as a bias-check in the ConceptNet build process, and can be found in the

`conceptnet5.vectors.evaluation.bias` module. I'm interested in expanding this to more ethnic backgrounds, which may require looking at surnames and not just given names.

Here are the lists:

```
NAMES_BY_ETHNICITY = {
    # The first two lists are from the Caliskan et al. appendix descr
ibing the
    # Word Embedding Association Test.
    'White': [
        'Adam', 'Chip', 'Harry', 'Josh', 'Roger', 'Alan', 'Frank', 'I
an', 'Justin',
        'Ryan', 'Andrew', 'Fred', 'Jack', 'Matthew', 'Stephen', 'Brad
', 'Greg', 'Jed',
        'Paul', 'Todd', 'Brandon', 'Hank', 'Jonathan', 'Peter', 'Wilb
ur', 'Amanda',
        'Courtney', 'Heather', 'Melanie', 'Sara', 'Amber', 'Crystal',
 'Katie',
        'Meredith', 'Shannon', 'Betsy', 'Donna', 'Kristin', 'Nancy',
'Stephanie',
        'Bobbie-Sue', 'Ellen', 'Lauren', 'Peggy', 'Sue-Ellen', 'Colle
en', 'Emily',
        'Megan', 'Rachel', 'Wendy'
    ],

    'Black': [
        'Alonzo', 'Jamel', 'Lerone', 'Percell', 'Theo', 'Alphonse', '
Jerome',
        'Leroy', 'Rasaan', 'Torrance', 'Darnell', 'Lamar', 'Lionel',
'Rashaun',
        'Tyree', 'Deion', 'Lamont', 'Malik', 'Terrence', 'Tyrone', 'E
verol',
        'Lavon', 'Marcellus', 'Terryl', 'Wardell', 'Aiesha', 'Lashell
e', 'Nichelle',
        'Shereen', 'Temeka', 'Ebony', 'Latisha', 'Shaniqua', 'Tameish
a', 'Teretha',
        'Jasmine', 'Latonya', 'Shanise', 'Tanisha', 'Tia', 'Lakisha',
 'Latoya',
        'Sharise', 'Tashika', 'Yolanda', 'Lashandra', 'Malika', 'Shav
onn',
        'Tawanda', 'Yvette'
    ],
```

```
    # This list comes from statistics about common Hispanic-origin na
mes in the US.
    'Hispanic': [
        'Juan', 'José', 'Miguel', 'Luís', 'Jorge', 'Santiago', 'Matía
s', 'Sebastián',
        'Mateo', 'Nicolás', 'Alejandro', 'Samuel', 'Diego', 'Daniel',
 'Tomás',
        'Juana', 'Ana', 'Luisa', 'María', 'Elena', 'Sofía', 'Isabella
', 'Valentina',
        'Camila', 'Valeria', 'Ximena', 'Luciana', 'Mariana', 'Victori
a', 'Martina'
    ],

    # The following list conflates religion and ethnicity, I'm aware.
 So do given names.
    #
    # This list was cobbled together from searching baby-name sites f
or common Muslim names,
    # as spelled in English. I did not ultimately distinguish whether
 the origin of the name
    # is Arabic or Urdu or another language.
    #
    # I'd be happy to replace it with something more authoritative, g
iven a source.
    'Arab/Muslim': [
        'Mohammed', 'Omar', 'Ahmed', 'Ali', 'Youssef', 'Abdullah', 'Y
asin', 'Hamza',
        'Ayaan', 'Syed', 'Rishaan', 'Samar', 'Ahmad', 'Zikri', 'Rayya
n', 'Mariam',
        'Jana', 'Malak', 'Salma', 'Nour', 'Lian', 'Fatima', 'Ayesha',
 'Zahra', 'Sana',
        'Zara', 'Alya', 'Shaista', 'Zoya', 'Yasmin'
    ]
}
```

Now we'll use Pandas to make a table of these names, their predominant ethnic background, and the sentiment score we get for them:

```
def name_sentiment_table():
    frames = []
    for group, name_list in sorted(NAMES_BY_ETHNICITY.items()):
        lower_names = [name.lower() for name in name_list]
```

```
        sentiments = words_to_sentiment(lower_names)
        sentiments['group'] = group
        frames.append(sentiments)

    # Put together the data we got from each ethnic group into one bi
g table
    return pd.concat(frames)

name_sentiments = name_sentiment_table()
```

A sample of the data:

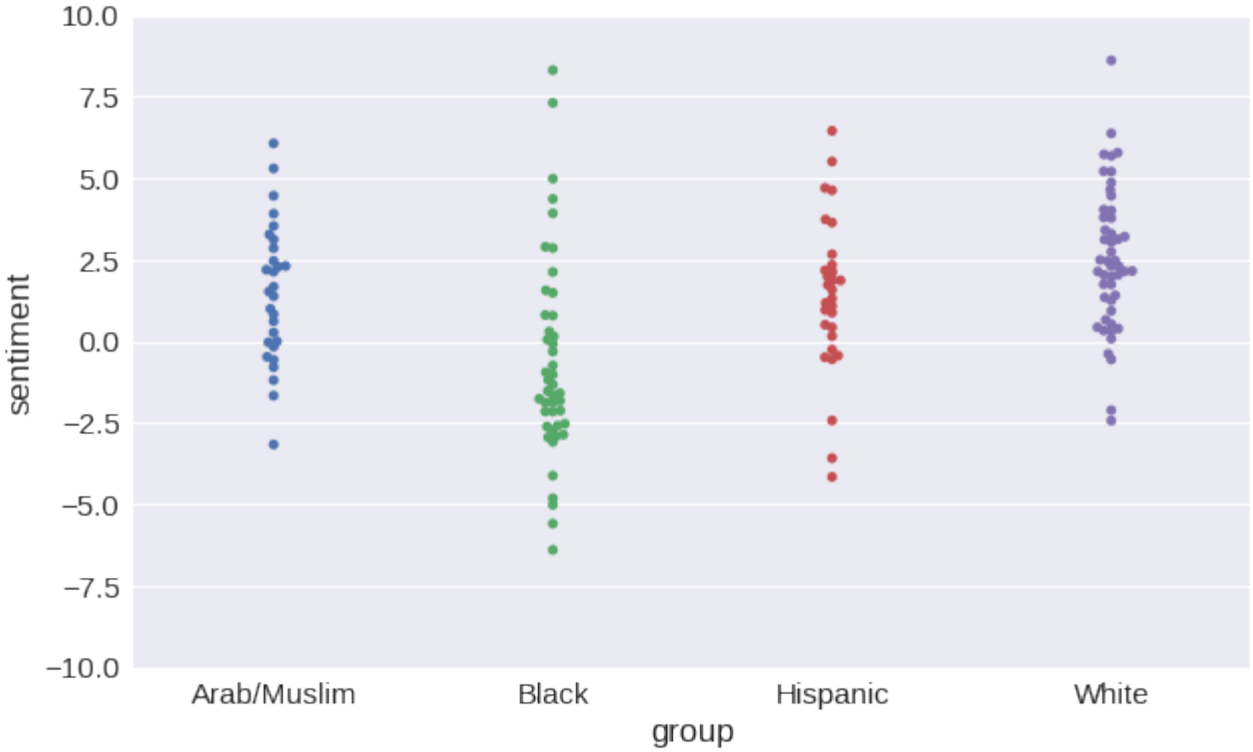
```
name_sentiments.ix[::25]
```

	sentiment	group
mohammed	0.834974	Arab/Muslim
alya	3.916803	Arab/Muslim
terryl	-2.858010	Black
josé	0.432956	Hispanic
luciana	1.086073	Hispanic
hank	0.391858	White
megan	2.158679	White

Now we can visualize the distribution of sentiment we get for each kind of name:

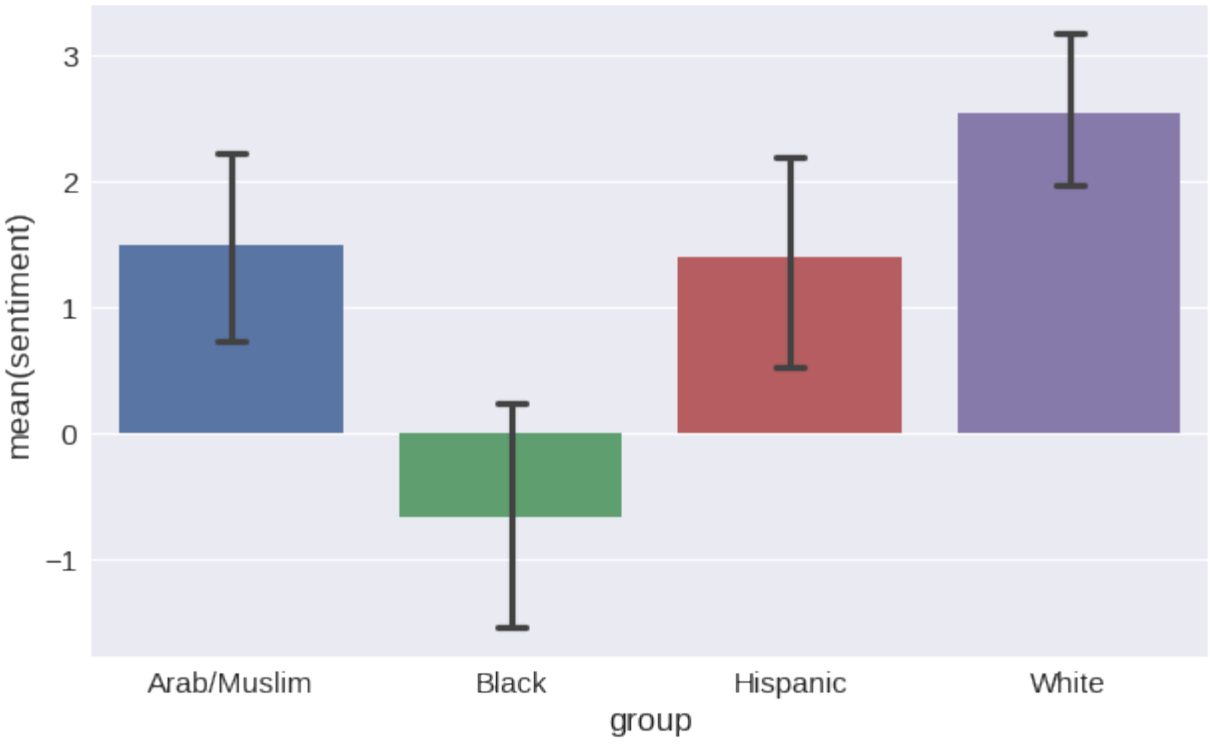
```
plot = seaborn.swarmplot(x='group', y='sentiment', data=name_sentimen
ts)
plot.set_ylim([-10, 10])
```

```
(-10, 10)
```



We can see that as a bar-plot, too, showing the 95% confidence intervals of the means.

```
plot = seaborn.barplot(x='group', y='sentiment', data=name_sentiments
, capsize=.1)
```



And finally we can break out the serious statistical machinery, using the statsmodels package, to tell us how big of an effect this is (along with a bunch of other statistics).

```
ols_model = statsmodels.formula.api.ols('sentiment ~ group', data=nam
e_sentiments).fit()
ols_model.summary().tables[0]
```

OLS Regression Results

Dep. Variable:	sentiment	**R-squared:**	0.208
Model:	OLS	**Adj. R-squared:**	0.192
Method:	Least Squares	**F-statistic:**	13.04
Date:	Thu, 13 Jul 2017	**Prob (F-statistic):**	1.31e-07
Time:	11:31:17	**Log-Likelihood:**	-356.78
No. Observations:	153	**AIC:**	721.6
Df Residuals:	149	**BIC:**	733.7
Df Model:	3		
Covariance Type:	nonrobust		

The F-statistic is the ratio of the variation between groups to the variation within groups, which we can take as a measure of overall ethnic bias.

The probability, right below that, is the probability that we would see this high of an F-statistic given the null hypothesis: that is, given data where there was no difference between ethnicities. The probability is very, very low. If this were a paper, we'd get to call the result "highly statistically significant".

Out of all these numbers, the F-value is the one we really want to improve. A lower F-value is better.

```
ols_model.fvalue
```

```
13.041597745167659
```

Step 7: Trying different data

Now that we have the ability to measure prejudicial badness in our word vectors, let's try to improve it. To do so, we'll want to repeat a bunch of things that so far we just ran as individual steps in this Python notebook.

If I were writing good, maintainable code, I wouldn't have been using global variables like `model` and `embeddings`. But writing ad-hoc spaghetti research code let us look at what we were doing at every step and learn from it, so there's something to be said for that. Let's re-use what we can, and at least define a function for redoing some of these steps:

```
def retrain_model(new_embs):
    """
    Repeat the steps above with a new set of word embeddings.
    """
    global model, embeddings, name_sentiments
    embeddings = new_embs
    pos_vectors = embeddings.loc[pos_words].dropna()
    neg_vectors = embeddings.loc[neg_words].dropna()
    vectors = pd.concat([pos_vectors, neg_vectors])
    targets = np.array([1 for entry in pos_vectors.index] + [-1 for e
ntry in neg_vectors.index])
    labels = list(pos_vectors.index) + list(neg_vectors.index)

    train_vectors, test_vectors, train_targets, test_targets, train_l
abels, test_labels = \
        train_test_split(vectors, targets, labels, test_size=0.1, ran
dom_state=0)
```

```
    model = SGDClassifier(loss='log', random_state=0, n_iter=100)
    model.fit(train_vectors, train_targets)

    accuracy = accuracy_score(model.predict(test_vectors), test_targe
ts)
    print("Accuracy of sentiment: {:.2%}".format(accuracy))

    name_sentiments = name_sentiment_table()
    ols_model = statsmodels.formula.api.ols('sentiment ~ group', data
=name_sentiments).fit()
    print("F-value of bias: {:.3f}".format(ols_model.fvalue))
    print("Probability given null hypothesis: {:.3}".format(ols_model
.f_pvalue))

    # Show the results on a swarm plot, with a consistent Y-axis
    plot = seaborn.swarmplot(x='group', y='sentiment', data=name_sent
iments)
    plot.set_ylim([-10, 10])
```

Trying word2vec

You may think this is a problem that only GloVe has. If the system weren't trained on all of the Common Crawl (which contains lots of unsavory sites and like 20 copies of Urban Dictionary), maybe it wouldn't have gone bad. What about good old word2vec, trained on Google News?

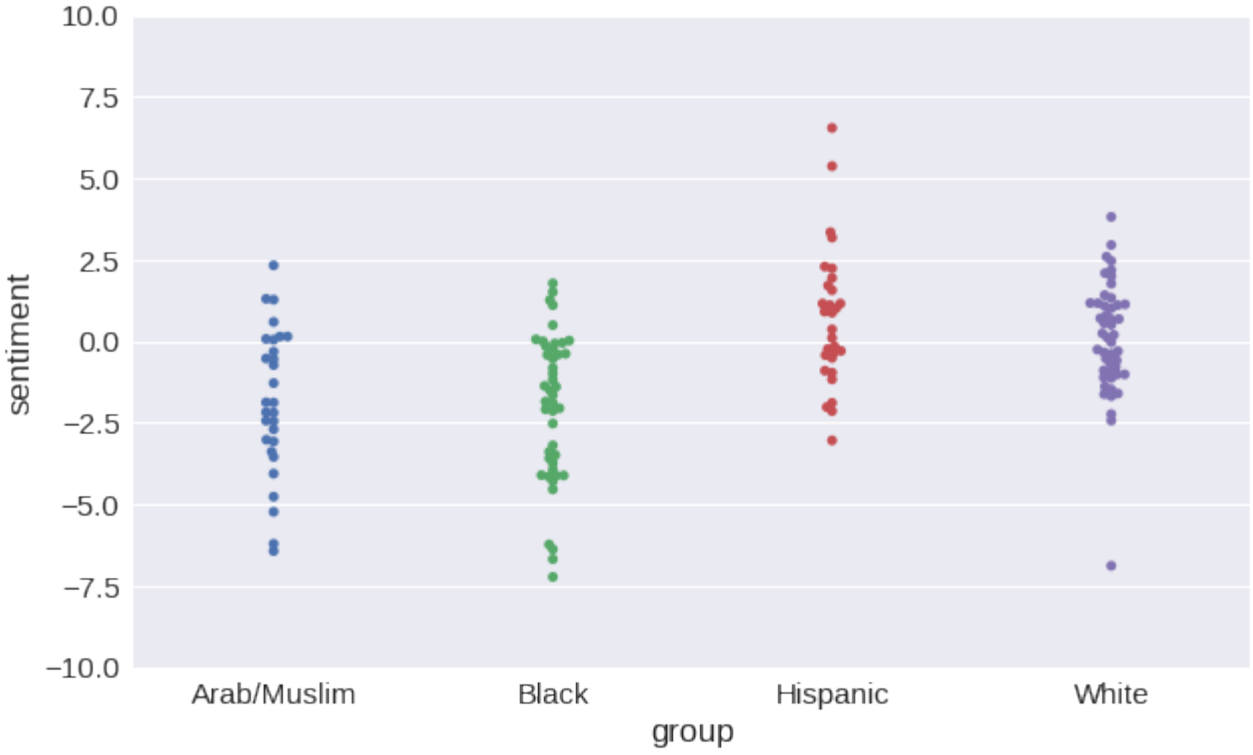
The most authoritative source for the word2vec data seems to be this file on Google Drive. Download it and save it as `data/word2vec-googlenews-300.bin.gz`.

```
# Use a ConceptNet function to load word2vec into a Pandas frame from
 its binary format
from conceptnet5.vectors.formats import load_word2vec_bin
w2v = load_word2vec_bin('data/word2vec-googlenews-300.bin.gz', nrows=
2000000)

# word2vec is case-sensitive, so case-fold its labels
w2v.index = [label.casefold() for label in w2v.index]
```

```
# Now we have duplicate labels, so drop the later (lower-frequency) o
ccurrences of the same label
w2v = w2v.reset_index().drop_duplicates(subset='index', keep='first')
.set_index('index')
retrain_model(w2v)
```

```
Accuracy of sentiment: 94.30%
F-value of bias: 15.573
Probability given null hypothesis: 7.43e-09
```



So: word2vec is even worse. With an F-value over 15, it has even larger differences in sentiment between groups.

In retrospect, expecting *news* to be safe from algorithmic bias was rather a lot to hope for.

Trying ConceptNet Numberbatch

Now I can finally get to discussing my own word-embedding project.

ConceptNet, the knowledge graph I work on with word-embedding

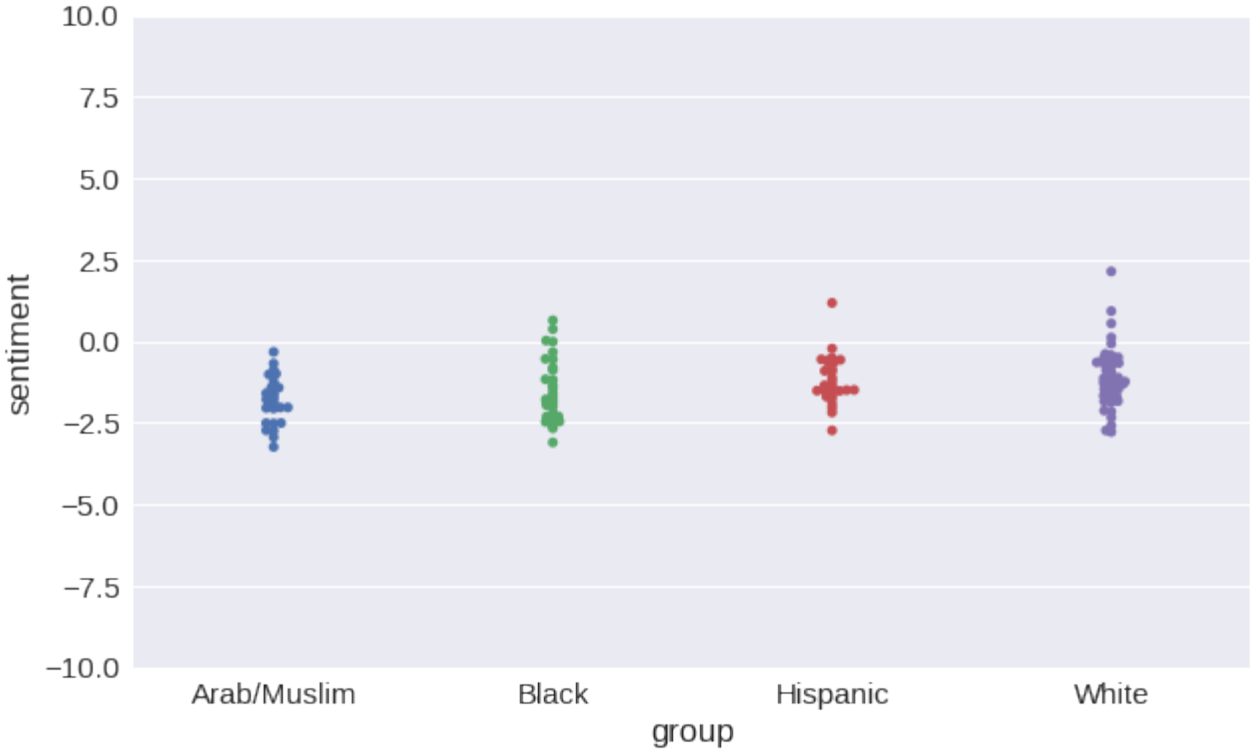
features built in, has a training step that adjusts the embeddings to identify and remove some sources of algorithmic racism and sexism. This step is based on Bolukbasi et al.'s "Debiasing Word Embeddings", and generalized to address multiple forms of prejudice at once. As far as I know, we're the only semantic system that has anything of the sort built in.

From time to time, we export pre-computed vectors from ConceptNet, a release we give the name ConceptNet Numberbatch. The April 2017 release was the first to include this de-biasing step, so let's load its English vectors and retrain our sentiment model with them.

Download `numberbatch-en-17.04b.txt.gz`, save it in the `data/` directory, and retrain the model:

```
retrain_model(load_embeddings('data/numberbatch-en-17.04b.txt'))
```

```
Accuracy of sentiment: 97.46%
F-value of bias: 3.805
Probability given null hypothesis: 0.0118
```



So have we entirely fixed the problem by switching to ConceptNet Numberbatch? Can we stop worrying about algorithmic racism? **No.**

Have we made the problem a lot smaller? **Definitely.**

The ranges of sentiments overlap a lot more than they did in the word vectors that came directly from GloVe or word2vec. The F-value is less than a third of what it was for GloVe, and a quarter of what it was for word2vec. And in general, we see much smaller differences in sentiment that come from comparing different given names, which is what we'd hope for, because names really shouldn't matter to the task of sentiment analysis.

But there is still a small correlation. Maybe I could have picked some data or training parameters that made the problem look completely solved. That would have been a bad move, because the problem *isn't* completely solved. There are more causes of

algorithmic racism than the ones we have identified and compensated for in ConceptNet. But this is a good start.

There is no trade-off

Note that the accuracy of sentiment prediction went *up* when we switched to ConceptNet Numberbatch.

Some people expect that fighting algorithmic racism is going to come with some sort of trade-off. There's no trade-off here. You can have data that's better and less racist. You can have data that's better *because* it's less racist. There was never anything "accurate" about the overt racism that word2vec and GloVe learned.

Other approaches

This is of course only one way to do sentiment analysis. All the steps we used are common, but you probably object that you wouldn't do it that way. But if you have your own process, I urge you to see if your process is encoding prejudices and biases in the model it learns.

Instead of or in addition to changing your source of word vectors, you could try to fix this problem in the output directly. It may help, for example, to build a stronger model of whether sentiment should be assigned to words at all, designed to specifically exclude names and groups of people.

You could abandon the idea of inferring sentiment for words, and only count the sentiment of words that appear exactly in the list. This is perhaps the most common form of sentiment analysis —

the kind that includes no machine learning at all. Its results will be no more biased than whoever made the list. But the lack of machine learning means that this approach has low recall, and the only way to adapt it to your data set is to edit the list manually.

As a hybrid approach, you could produce a large number of inferred sentiments for words, and have a human annotator patiently look through them, making a list of exceptions whose sentiment should be set to 0. The downside of this is that it's extra work; the upside is that you take the time to actually see what your data is doing. And that's something that I think should happen more often in machine learning anyway.

NLP fairness Tutorials

Previous post Next post

Comments