06_sentiment_analysis_yelp

September 29, 2021

1 Text classification and sentiment analysis: Yelp Reviews

Once text data has been converted into numerical features using the natural language processing techniques discussed in the previous sections, text classification works just like any other classification task.

In this notebook, we will apply these preprocessing technique to Yelp business reviews to classify them by review scores and sentiment polarity. More specifically, we will apply sentiment analysis to the significantly larger Yelp business review dataset with five outcome classes.

1.1 Imports

[1]:
```
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
%matplotlib inline

from pathlib import Path
import json
from time import time

import numpy as np
import pandas as pd

from scipy import sparse

# spacy, textblob and nltk for language processing
from textblob import TextBlob

# sklearn for feature extraction & modeling
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix
import joblib

import lightgbm as lgb
```

```python
# Visualization
import matplotlib.pyplot as plt
import seaborn as sns
```

[3]:
```python
sns.set_style('white')
np.random.seed(42)
```

1.2 Yelp Challenge: business reviews dataset

1.2.1 Load Data

Follow the instructions to create the dataset.

[4]:
```python
data_dir = Path('..', 'data', 'yelp')
```

[5]:
```python
yelp_reviews = pd.read_parquet(data_dir / 'user_reviews.parquet')
```

[6]:
```python
yelp_reviews.info(null_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 8021122 entries, 0 to 8021121
Data columns (total 25 columns):
 #   Column              Non-Null Count    Dtype
---  ------              --------------    -----
 0   stars               8021122 non-null  float64
 1   useful              8021122 non-null  int64
 2   funny               8021122 non-null  int64
 3   cool                8021122 non-null  int64
 4   text                8021122 non-null  object
 5   year                8021122 non-null  int64
 6   month               8021122 non-null  int64
 7   review_count        8021122 non-null  int64
 8   useful_user         8021122 non-null  int64
 9   funny_user          8021122 non-null  int64
 10  cool_user           8021122 non-null  int64
 11  fans                8021122 non-null  int64
 12  average_stars       8021122 non-null  float64
 13  compliment_hot      8021122 non-null  int64
 14  compliment_more     8021122 non-null  int64
 15  compliment_profile  8021122 non-null  int64
 16  compliment_cute     8021122 non-null  int64
 17  compliment_list     8021122 non-null  int64
 18  compliment_note     8021122 non-null  int64
 19  compliment_plain    8021122 non-null  int64
 20  compliment_cool     8021122 non-null  int64
 21  compliment_funny    8021122 non-null  int64
 22  compliment_writer   8021122 non-null  int64
 23  compliment_photos   8021122 non-null  int64
```

```
 24  member_yrs          8021122 non-null  int64
dtypes: float64(2), int64(22), object(1)
memory usage: 1.6+ GB
```

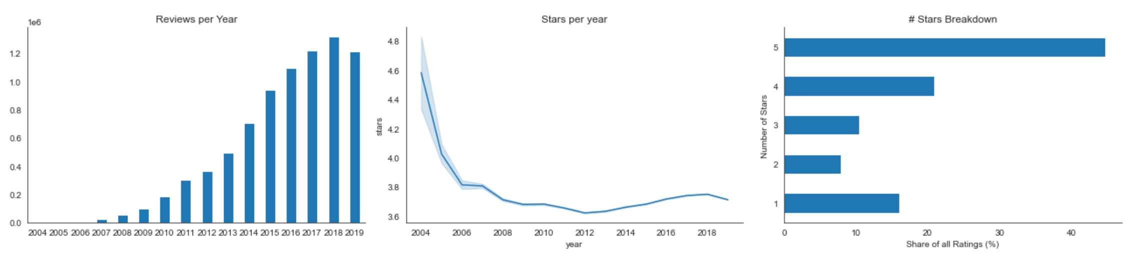
1.2.2 Explore data

[13]:
```python
yelp_dir = Path('results', 'yelp')

text_features_dir = yelp_dir / 'data'
if not text_features_dir.exists():
    text_features_dir.mkdir(exist_ok=True, parents=True)
```

The following figure shows the number of reviews and the average number of stars per year.

Reviews & Stars by Year

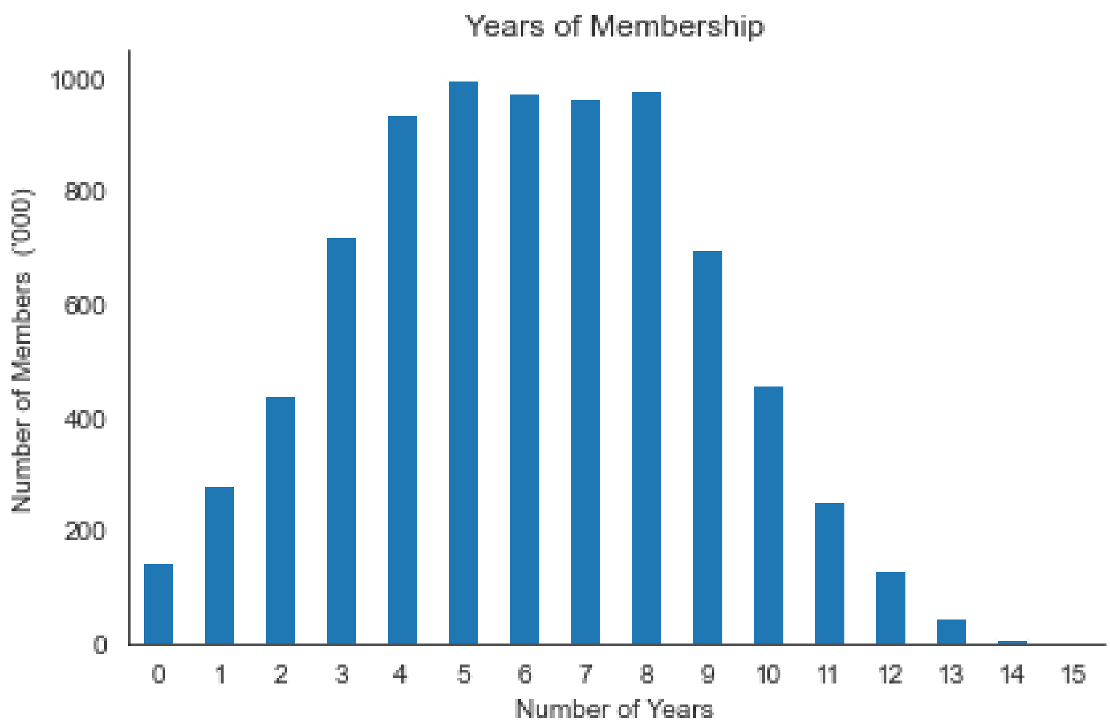
[9]:
```python
fig, axes = plt.subplots(ncols=3, figsize=(18, 4))
yelp_reviews.year.value_counts().sort_index().plot.bar(title='Reviews per Year', ax=axes[0], rot=0);
sns.lineplot(x='year', y='stars', data=yelp_reviews, ax=axes[1])
axes[1].set_title('Stars per year')

stars_dist = yelp_reviews.stars.value_counts(normalize=True).sort_index().mul(100)
stars_dist.index = stars_dist.index.astype(int)
stars_dist.plot.barh(title='# Stars Breakdown', ax=axes[2])
axes[2].set_xlabel('Share of all Ratings (%)')
axes[2].set_ylabel('Number of Stars');

sns.despine()
fig.tight_layout();
```



Years of Membership Breakdown

[10]:
```python
ax = yelp_reviews.member_yrs.value_counts().div(1000).sort_index().plot.bar(title='Years of Membership',
                                                                            rot=0)
```

```
ax.set_xlabel('Number of Years')
ax.set_ylabel("Number of Members  ('000)")
sns.despine()
plt.tight_layout()
```



1.2.3 Create train-test split

[11]:
```
train = yelp_reviews[yelp_reviews.year < 2019].sample(frac=.25)
test = yelp_reviews[yelp_reviews.year == 2019]
```

[12]:
```
print(f'# Training Obs: {len(train):,.0f} | # Test Obs: {len(test):,.0f}')
```

```
# Training Obs: 1,701,322 | # Test Obs: 1,215,836
```

[14]:
```
train.to_parquet(text_features_dir / 'train.parquet')
test.to_parquet(text_features_dir / 'test.parquet')
```

[21]:
```
del yelp_reviews
```

Reload stored data

[64]:
```
train = pd.read_parquet(text_features_dir / 'train.parquet')
test = pd.read_parquet(text_features_dir / 'test.parquet')
```

1.3 Create Yelp review document-term matrix

[16]:
```
vectorizer = CountVectorizer(stop_words='english', ngram_range=(1, 2),␣
 ↪max_features=10000)
train_dtm = vectorizer.fit_transform(train.text)
train_dtm
```

[16]:
```
<1701322x10000 sparse matrix of type '<class 'numpy.int64'>'
        with 75720724 stored elements in Compressed Sparse Row format>
```

[17]:
```
sparse.save_npz(text_features_dir / 'train_dtm', train_dtm)
```

[18]:
```
test_dtm = vectorizer.transform(test.text)
sparse.save_npz(text_features_dir / 'test_dtm', test_dtm)
```

1.3.1 Reload stored data

[7]:
```
train_dtm = sparse.load_npz(text_features_dir / 'train_dtm.npz')
test_dtm = sparse.load_npz(text_features_dir / 'test_dtm.npz')
```

1.4 Combine non-text features with the document-term matrix

The dataset contains various numerical features. The vectorizers produce scipy.sparse matrices. To combine the vectorized text data with other features, we need to first convert these to sparse matrices as well; many sklearn objects and other libraries like lightgbm can handle these very memory-efficient data structures. Converting the sparse matrix to a dense numpy array risks memory overflow.

Most variables are categorical so we use one-hot encoding since we have a fairly large dataset to accommodate the increase in features.

We convert the encoded numerical features and combine them with the document-term matrix:

1.4.1 One-hot-encoding

[19]:
```
df = pd.concat([train.drop(['text', 'stars'], axis=1).assign(source='train'),
                test.drop(['text', 'stars'], axis=1).assign(source='test')])
```

[20]:
```
uniques = df.nunique()
binned = pd.concat([(df.loc[:, uniques[uniques > 20].index]
                     .apply(pd.qcut, q=10, labels=False, duplicates='drop')),
                    df.loc[:, uniques[uniques <= 20].index]], axis=1)
binned.info(null_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 2917158 entries, 4906334 to 8021121
Data columns (total 24 columns):
 #   Column              Non-Null Count    Dtype
```

```
---  ------              --------------   -----
 0   useful              2917158 non-null  int64
 1   funny               2917158 non-null  int64
 2   cool                2917158 non-null  int64
 3   review_count        2917158 non-null  int64
 4   useful_user         2917158 non-null  int64
 5   funny_user          2917158 non-null  int64
 6   cool_user           2917158 non-null  int64
 7   fans                2917158 non-null  int64
 8   average_stars       2917158 non-null  int64
 9   compliment_hot      2917158 non-null  int64
 10  compliment_more     2917158 non-null  int64
 11  compliment_profile  2917158 non-null  int64
 12  compliment_cute     2917158 non-null  int64
 13  compliment_list     2917158 non-null  int64
 14  compliment_note     2917158 non-null  int64
 15  compliment_plain    2917158 non-null  int64
 16  compliment_cool     2917158 non-null  int64
 17  compliment_funny    2917158 non-null  int64
 18  compliment_writer   2917158 non-null  int64
 19  compliment_photos   2917158 non-null  int64
 20  year                2917158 non-null  int64
 21  month               2917158 non-null  int64
 22  member_yrs          2917158 non-null  int64
 23  source              2917158 non-null  object
dtypes: int64(23), object(1)
memory usage: 556.4+ MB
```

[22]:
```
dummies = pd.get_dummies(binned,
                         columns=binned.columns.drop('source'),
                         drop_first=True)
dummies.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 2917158 entries, 4906334 to 8021121
Columns: 111 entries, source to member_yrs_15
dtypes: object(1), uint8(110)
memory usage: 350.5+ MB
```

[23]:
```
train_dummies = dummies[dummies.source=='train'].drop('source', axis=1)
train_dummies.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1701322 entries, 4906334 to 4178053
Columns: 110 entries, useful_1 to member_yrs_15
dtypes: uint8(110)
memory usage: 191.5 MB
```

1.4.2 Train set

[24]:
```
# Cast other feature columns to float and convert to a sparse matrix.
train_numeric = sparse.csr_matrix(train_dummies.astype(np.uint8))
train_numeric.shape
```

[24]: (1701322, 110)

[25]:
```
# Combine sparse matrices.
train_dtm_numeric = sparse.hstack((train_dtm, train_numeric))
train_dtm_numeric.shape
```

[25]: (1701322, 10110)

[26]:
```
sparse.save_npz(text_features_dir / 'train_dtm_numeric',
                train_dtm_numeric)
```

1.4.3 Repeat for test set

[27]:
```
test_dummies = dummies[dummies.source=='test'].drop('source', axis=1)
test_numeric = sparse.csr_matrix(test_dummies.astype(np.int8))
test_dtm_numeric = sparse.hstack((test_dtm, test_numeric))
test_dtm_numeric.shape
```

[27]: (1215836, 10110)

[28]:
```
sparse.save_npz(text_features_dir / 'test_dtm_numeric', test_dtm_numeric)
```

1.4.4 Reload stored data

[ ]:
```
train_dtm_numeric = sparse.load_npz(text_features_dir / 'train_dtm_numeric.npz')
test_dtm_numeric = sparse.load_npz(text_features_dir / 'test_dtm_numeric.npz')
```

1.5 Benchmark Accuracy

[29]:
```
accuracy, runtime = {}, {}
predictions = test[['stars']].copy()
```

Using the most frequent number of stars (=5) to predict the test set achieve an accuracy close to 51%:

[30]:
```
naive_prediction = np.full_like(predictions.stars,
                                fill_value=train.stars.mode().iloc[0])
```

[31]:
```
naive_benchmark = accuracy_score(predictions.stars, naive_prediction)
```

[32]:
```
naive_benchmark
```

[32]: 0.5117779042568241

1.6 Model Evaluation Helper

[33]:
```
def evaluate_model(model, X_train, X_test, name, store=False):
    start = time()
    model.fit(X_train, train.stars)
    runtime[name] = time() - start
    predictions[name] = model.predict(X_test)
    accuracy[result] = accuracy_score(test.stars, predictions[result])
    if store:
        joblib.dump(model, f'results/{result}.joblib')
```

1.7 Multiclass Naive Bayes

[34]:
```
nb = MultinomialNB()
```

1.7.1 Text Features

Next, we train a Naive Bayes classifier using a document-term matrix produced by the CountVectorizer with default settings.

[35]:
```
result = 'nb_text'
```

[36]:
```
evaluate_model(nb, train_dtm, test_dtm, result, store=False)
```

Accuracy The prediction produces 64.4% accuracy on the test set, a 24.2% improvement over the benchmark:

[37]:
```
accuracy[result]
```

[37]: 0.6520747864021135

Confusion Matrix

[38]:
```
stars = index = list(range(1, 6))
pd.DataFrame(confusion_matrix(test.stars,
                              predictions[result]),
             columns=stars,
             index=stars)
```

[38]:

	1	2	3	4	5
1	178787	42369	8419	3193	4295
2	26795	30224	18533	4445	3025
3	12420	17233	37337	21719	6319
4	7907	5608	21599	100153	43218
5	31323	3766	6762	134072	446315

1.7.2 Text & Numeric Features

[39]:
```
result = 'nb_combined'
```

[40]:
```
evaluate_model(nb, train_dtm_numeric, test_dtm_numeric, result, store=False)
```

Accuracy

[41]:
```
accuracy[result]
```

[41]:
```
0.6739017433272251
```

1.8 Multinomial Logistic Regression

Logistic regression also provides a multinomial training option that is faster and more accurate than the one-vs-all implementation. We use the lbfgs solver (see sklearn documentation for details).

[42]:
```
Cs = np.logspace(-5, 5, 11)
```

1.8.1 Text Features

[45]:
```
log_reg_text_accuracy = {}
log_reg_text_runtime = []
for i, C in enumerate(Cs):
    start = time()
    model = LogisticRegression(C=C,
                               multi_class='multinomial',
                               solver='lbfgs')

    model.fit(train_dtm, train.stars)
    log_reg_text_runtime.append(time() - start)
    log_reg_text_accuracy[C] = accuracy_score(test.stars,
                                              model.predict(test_dtm))

    print(f'{C:12.5f}: {log_reg_text_runtime[i]:.2f}s |␣
 ↪{log_reg_text_accuracy[C]:.2%}', flush=True)
```

```
     0.00001: 34.93s | 62.02%
     0.00010: 74.01s | 70.89%
     0.00100: 126.02s | 73.95%
     0.01000: 122.22s | 74.85%
     0.10000: 125.65s | 74.80%
     1.00000: 130.17s | 74.83%
    10.00000: 126.39s | 74.80%
   100.00000: 125.82s | 74.81%
  1000.00000: 122.73s | 74.83%
 10000.00000: 123.29s | 74.80%
100000.00000: 126.80s | 74.80%
```

[46]:
```
pd.Series(log_reg_text_accuracy).to_csv(yelp_dir / 'logreg_text.csv')
```

[47]:
```
accuracy['lr_text'] = pd.Series(log_reg_text_accuracy).max()
runtime['lr_text'] = np.mean(log_reg_text_runtime)
```

1.8.2 Combined Features

[50]:
```
log_reg_comb_accuracy = {}
log_reg_comb_runtime = []
for i, C in enumerate(Cs):
    start = time()
    model = LogisticRegression(C=C,
                               multi_class='multinomial',
                               solver='lbfgs')

    model.fit(train_dtm_numeric, train.stars)
    log_reg_comb_runtime.append(time() - start)
    log_reg_comb_accuracy[C] = accuracy_score(test.stars,
                                              model.predict(test_dtm_numeric))

    print(f'{C:12.5f}: {log_reg_comb_runtime[i]:.2f}s |␣
 ↪{log_reg_comb_accuracy[C]:.2%}', flush=True)
```

```
     0.00001: 55.26s | 63.98%
     0.00010: 99.05s | 72.94%
     0.00100: 137.62s | 75.12%
     0.01000: 139.11s | 75.55%
     0.10000: 140.26s | 75.28%
     1.00000: 138.90s | 75.32%
    10.00000: 135.13s | 75.35%
   100.00000: 137.76s | 75.32%
  1000.00000: 139.09s | 75.36%
 10000.00000: 135.01s | 75.37%
100000.00000: 134.73s | 75.38%
```

[51]:
```
pd.Series(log_reg_comb_accuracy).to_csv(yelp_dir / 'logreg_combined.csv')
```

[52]:
```
accuracy['lr_comb'] = pd.Series(log_reg_comb_accuracy).max()
runtime['lr_comb'] = np.mean(log_reg_comb_runtime)
```

1.9 Gradient Boosting

For illustration, we also train a lightgbm Gradient Boosting tree ensemble with default settings and multiclass objective.

[65]:
```
lgb_train = lgb.Dataset(data=train_dtm_numeric.tocsr().astype(np.float32),
                        label=train.stars.sub(1),
```

```
                             categorical_feature=list(range(train_dtm_numeric.
↪shape[1])))
```

[66]:
```
lgb_test = lgb.Dataset(data=test_dtm_numeric.tocsr().astype(np.float32),
                       label=test.stars.sub(1),
                       reference=lgb_train)
```

[67]:
```
param = {'objective': 'multiclass',
         'metrics': ['multi_error'],
         'num_class': 5}
```

[68]:
```
booster = lgb.train(params=param,
                    train_set=lgb_train,
                    num_boost_round=2000,
                    early_stopping_rounds=25,
                    valid_sets=[lgb_train, lgb_test],
                    verbose_eval=25)
```

```
Training until validation scores don't improve for 25 rounds
[25]    training's multi_error: 0.405759        valid_1's multi_error: 0.320156
[50]    training's multi_error: 0.370905        valid_1's multi_error: 0.297562
[75]    training's multi_error: 0.353564        valid_1's multi_error: 0.285789
[100]   training's multi_error: 0.34278 valid_1's multi_error: 0.278771
[125]   training's multi_error: 0.335184        valid_1's multi_error: 0.273733
[150]   training's multi_error: 0.329293        valid_1's multi_error: 0.269682
[175]   training's multi_error: 0.324728        valid_1's multi_error: 0.266682
[200]   training's multi_error: 0.320829        valid_1's multi_error: 0.264156
[225]   training's multi_error: 0.317401        valid_1's multi_error: 0.262016
[250]   training's multi_error: 0.314696        valid_1's multi_error: 0.259945
[275]   training's multi_error: 0.312185        valid_1's multi_error: 0.258219
[300]   training's multi_error: 0.3099  valid_1's multi_error: 0.256977
[325]   training's multi_error: 0.307957        valid_1's multi_error: 0.25591
[350]   training's multi_error: 0.306137        valid_1's multi_error: 0.254868
[375]   training's multi_error: 0.304627        valid_1's multi_error: 0.254025
[400]   training's multi_error: 0.302983        valid_1's multi_error: 0.253086
[425]   training's multi_error: 0.30154 valid_1's multi_error: 0.252281
[450]   training's multi_error: 0.300309        valid_1's multi_error: 0.251648
[475]   training's multi_error: 0.299089        valid_1's multi_error: 0.250989
[500]   training's multi_error: 0.297708        valid_1's multi_error: 0.250408
[525]   training's multi_error: 0.296714        valid_1's multi_error: 0.249912
[550]   training's multi_error: 0.295725        valid_1's multi_error: 0.249531
[575]   training's multi_error: 0.294649        valid_1's multi_error: 0.24918
[600]   training's multi_error: 0.293748        valid_1's multi_error: 0.248822
[625]   training's multi_error: 0.292762        valid_1's multi_error: 0.248478
[650]   training's multi_error: 0.291932        valid_1's multi_error: 0.248306
[675]   training's multi_error: 0.291082        valid_1's multi_error: 0.248
[700]   training's multi_error: 0.290352        valid_1's multi_error: 0.247695
```

```
[725]	training's multi_error: 0.289579	valid_1's multi_error: 0.247381
[750]	training's multi_error: 0.288837	valid_1's multi_error: 0.247131
[775]	training's multi_error: 0.288094	valid_1's multi_error: 0.246972
[800]	training's multi_error: 0.287377	valid_1's multi_error: 0.247014
Early stopping, best iteration is:
[776]	training's multi_error: 0.288074	valid_1's multi_error: 0.246952
```

[69]:
```
booster.save_model((yelp_dir / 'lgb_model.txt').as_posix());
```

[70]:
```
y_pred_class = booster.predict(test_dtm_numeric.astype(float))
```

The basic settings did not improve over the multinomial logistic regression, but further parameter tuning remains an unused option.

[71]:
```
accuracy['lgb_comb'] = accuracy_score(test.stars, y_pred_class.argmax(1) + 1)
```

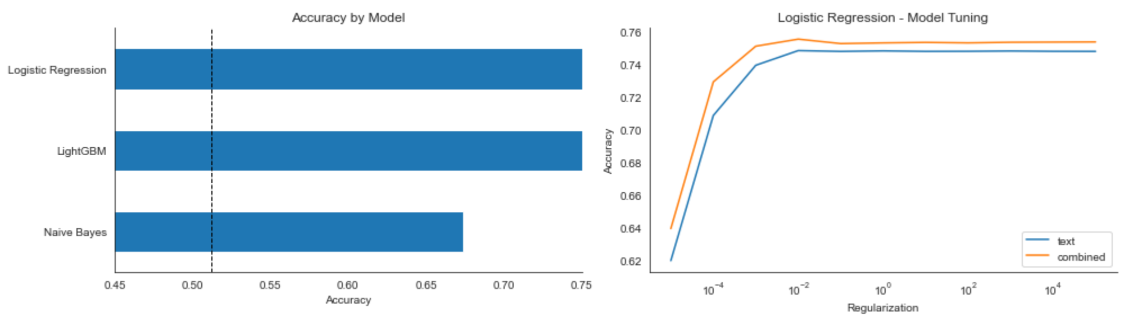
1.10 Comparison

[78]:
```
model_map = {'nb_combined': 'Naive Bayes',
             'lr_comb': 'Logistic Regression',
             'lgb_comb': 'LightGBM'}
```

[97]:
```
accuracy_ = {model_map[k]: v for k, v in accuracy.items() if model_map.get(k)}
```

[98]:
```
log_reg_text = pd.read_csv(yelp_dir / 'logreg_text.csv',
                           index_col=0,
                           squeeze=True)
log_reg_combined = pd.read_csv(yelp_dir / 'logreg_combined.csv',
                           index_col=0,
                           squeeze=True)
```

[101]:
```
fig, axes = plt.subplots(ncols=2, figsize=(14, 4))
pd.Series(accuracy_).sort_values().plot.barh(
    ax=axes[0], xlim=(.45, .75), title='Accuracy by Model')
axes[0].axvline(naive_benchmark, ls='--', lw=1, c='k')


log_reg = (log_reg_text.to_frame('text')
           .join(log_reg_combined.to_frame('combined')))
log_reg.plot(logx=True,
             ax=axes[1],
             title='Logistic Regression - Model Tuning')
axes[1].set_xlabel('Regularization')
axes[1].set_ylabel('Accuracy')
axes[0].set_xlabel('Accuracy')
sns.despine()
fig.tight_layout()
```



1.11 Textblob for Sentiment Analysis

[84]:
```
sample_review = train.text.sample(1).iloc[0]
print(sample_review)
```

```
It's good to see new restaurants in Henderson along Stephanie Street, and
judging by the packed parking lot a lot of people seem to like Miller's Ale
House. I can't say that I'm a fan. I'll agree with Ashley B's review that it's
better than Chili's and far better than suffering through Applebees, but it's
really just a ho-hum nothing special chain restaurant with TV's plastered
everywhere, and a boring menu of the same tired classics that can be easily
prepared by a bunch of distracted teenagers working the kitchen.

If you're hungry and the parking lot is not jammed and you can't decide on what
to eat at least you can shut your stomach up if you stop here. But if you want
something different or original remember this is a chain…you won't find what
you're craving here.

2 Stars…as in I was so underwhelmed by the place I almost fell asleep writing
this one. Yawn!
```

[85]:
```
# Polarity ranges from -1 (most negative) to 1 (most positive).
TextBlob(sample_review).sentiment.polarity
```

[85]: 0.11637265512265516

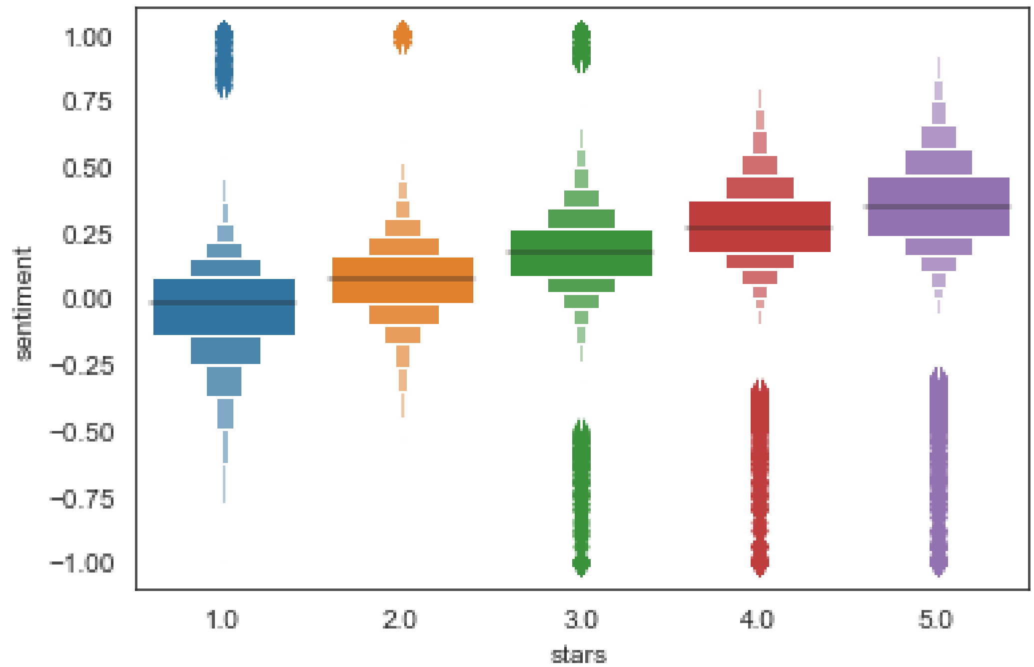
[86]:
```
# Define a function that accepts text and returns the polarity.
def detect_sentiment(text):
    return TextBlob(text).sentiment.polarity
```

[87]:
```
train['sentiment'] = train.text.apply(detect_sentiment)
```

[88]:
```
sample_reviews = train[['stars', 'text']].sample(100000)
```

[89]:
```
# Create a new DataFrame column for sentiment (Warning: SLOW!).
sample_reviews['sentiment'] = sample_reviews.text.apply(detect_sentiment)
```

[90]:
```
# Box plot of sentiment grouped by stars
sns.boxenplot(x='stars', y='sentiment', data=train);
```



[91]:
```
# Widen the column display.
pd.set_option('max_colwidth', 500)
```

[92]:
```
# Reviews with most negative sentiment
train[train.sentiment == -1].text.head()
```

[92]:
```
2495072                                                    TERRIBLE!! I Ordered
And They Just Cancelled My Order Because they "don't deliver to my hotel" no
notice or anything just cancelled!!! I Was Waiting For My\nOrder for about an
hour!
4807229
Don't bother to give them a call they did a no show for an estimate not even a
phone call to let me know horrible service
4379860
horrible, horrible, horrible.  food, service price.  it's an overpriced tourist
trap.  Pease do yourself a favor, go to in n out.
2941002
Well after my miserable experience they came with the rest of my furniture
today. All broken and cracked! Dump!!!!!!!!!!!!!!
```

```
2785241    This is probably the worst experience I have had at a restaurant.
Waited 10 minutes for even a server to come take my drink order.  Paid $15 for a
crappy breakfast.  \n\nI will never eat at this California pizza kitchen again.
Name: text, dtype: object
```

[93]:

```
# Negative sentiment in a 5-star review
train.loc[(train.stars == 5) & (train.sentiment < -0.3), 'text'].head(1)
```

[93]:

```
255057    Appointment set through Fidelity Home Warranty. Ken came out within a
couple of hours. The capacitor was bad on one of my units and replaced
immediately. Thank you Fidelity and Ken at Lee Collins Air.
Name: text, dtype: object
```

[94]:

```
# Positive sentiment in a 1-star review
train.loc[(train.stars == 1) & (train.sentiment > 0.5), 'text'].head(1)
```

[94]:

```
5373197    The food wasn't that great and the service was okay. We should have
just gone to our go to spot Baby Stacks. The steak and eggs had no flavor and
the potatoes were room temperature. I guess this is what happens when you want
breakfast for dinner. The bill was $46 which is fine but if the food was
actually good I don't think I would be as ann
Name: text, dtype: object
```

[95]:

```
# Reset the column display width.
pd.reset_option('max_colwidth')
```