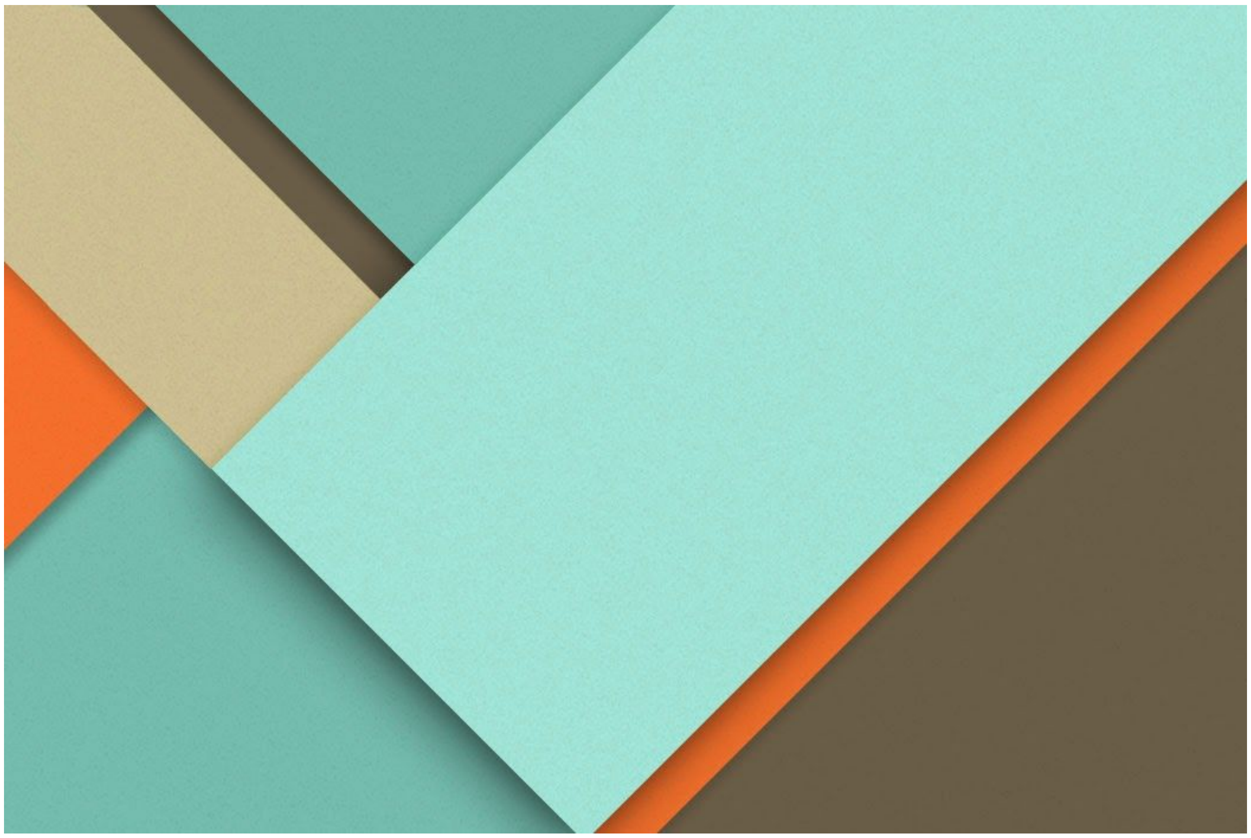


IMDB reviews- Text sentiment analysis

21.01.2020

Dotan Ishay and Noam Baron
Technion certified data science course
Rav Aluf David Elazar St 18, Tel Aviv-Yafo

Overview

Our task is to predict IMDB movies review sentiment - good VS bad using supervised dataset.

Link to the original project - https://www.kaggle.com/c/word2vec-nlp-tutorial/data

Goals

1. Gain experience in a 'real world' data science project.
2. Practice all (or most of) the machine learning techniques we've learned during the course, including traditional statistical learning and advanced deep learning methods in order to complete the task
3. Overcome the challenges of this specific task with creative out of the box solutions

Dataset

The labeled data set consists of 50,000 IMDB movie reviews, specially selected for sentiment analysis. The sentiment of reviews is binary, meaning the IMDB rating < 5 results in a sentiment score of 0, and rating >=7 have a sentiment score of 1.

As we can see our data is not biased. We have equal representation for both sentiments.

```
original_dataset.head()
         id  sentiment                                             review
0  "5814_8"          1  "With all this stuff going down at the moment ...
1  "2381_9"          1  "\"The Classic War of the Worlds\" by Timothy ...
2  "7759_3"          0  "The film starts with a manager (Nicholas Bell...
3  "3630_4"          0  "It must be assumed that those who praised thi...
4  "9495_8"          1  "Superbly trashy and wondrously unpretentious ...
```

```
original_dataset.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 25000 entries, 0 to 24999
Data columns (total 3 columns):
id           25000 non-null object
sentiment    25000 non-null int64
review       25000 non-null object
dtypes: int64(1), object(2)
memory usage: 586.1+ KB
```

Plan of work

- Running traditional machine learning classification models on the dataset with new features we will extract using nlp packages.
- Running the same models on features extracted by BoW and TF iDF .
- Running deep learning models - BERT, flair .

Dataset challenges

1. We have 25000 labeled reviews, from which we use 70% train and 30% test. Meaning our data for modeling is quite big - 17500 review.
2. Each review contains 2-20 sentences - we need to train on 350000 sentences worst case scenario. Our private computing machines are not capable to deal with this amount of data very easily.
3. Each review is very detailed, not very sammeried or preciced. A part of contain the “tone” of the review it contains lots of details about the movie itself, characters, direction, reference to similar movies, cynical reviews and “Noisy” reviews. There are no straight, sentiment driven reviews as we hope to work with like in an ideal world .

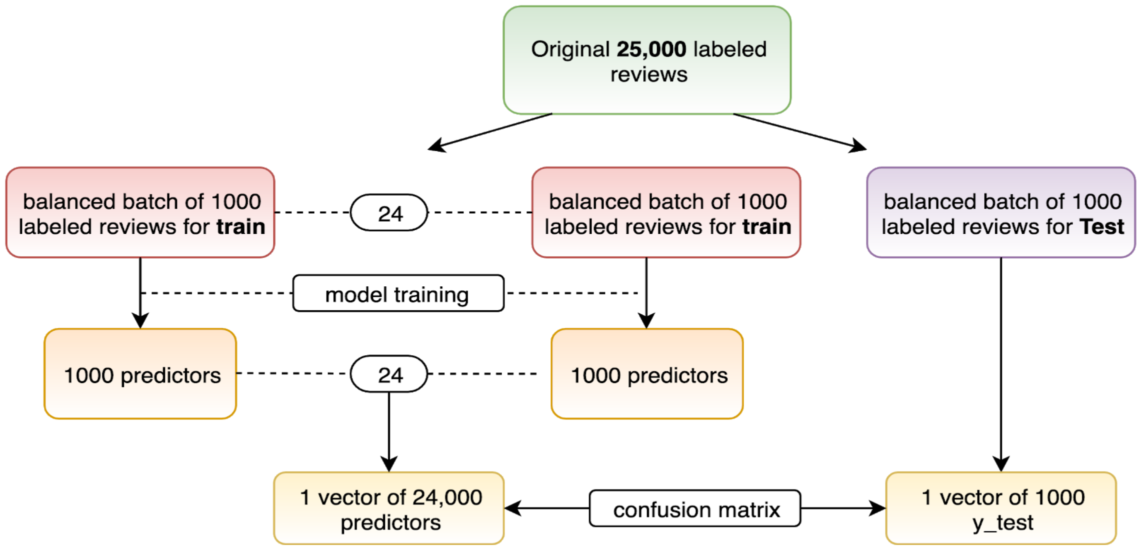
Models

PreProcessing

Due to the fact that our original dataset was very big, each nlp model takes a lot of time the run, even simple data cleaning.
Moreover, our personal computing machines are not built to run complex models in a reasonable run time without crash.

Thus, We've decided to divide our dataset **into 25 batches of 1000** reviews each batch, **keeping the target balanced**. Each model that we are using will only run on 1000 examples at a time. 1 of the 25 batches was always out 'test' set

After modeling we have 24 batches of 1000 predictors. We then concat them together so each model result will have predictor vector size 24000 predictors compared to test vector size 1000.


Original 25,000 labeled reviews
balanced batch of 1000 labeled reviews for train
24
balanced batch of 1000 labeled reviews for train
balanced batch of 1000 labeled reviews for Test
model training
1000 predictors
24
1000 predictors
1 vector of 24,000 predictors
confusion matrix
1 vector of 1000 y_test

EDA - feature extraction

Rule Based Methods - Sentiment scoring - TextBlob and vader (NLTK)

By using Rule-based library for sentiment analysis we were able to extract numerical features such as 'polarity score' and 'subjectivity' and use their values with simple decision rules to predict the target

```
from textblob import TextBlob
from nltk.sentiment.vader import SentimentIntensityAnalyzer

ssl._create_default_https_context = ssl._create_unverified_context
    nltk.download('vader_lexicon', quiet=True)

class TextBlobSentiment(RuleBasedClass):
   def __init__(self):
       super().__init__('TextBlob')

   @staticmethod
   def _score(row):
       return TextBlob(row[FieldsEnum.REVIEW]).sentiment.polarity


class VaderSentiment(RuleBasedClass):
   def __init__(self):
       super().__init__('Vader')

   def _generate_model(self):
       self.model = SentimentIntensityAnalyzer()
```

TextBlob, built on top of NLTK uses a sentiment lexicon (consisting of predefined words) to assign scores for each word, which are then averaged out using a weighted average to give an overall sentence sentiment score.Three scores: **'polarity', 'subjectivity' and 'intensity'** are calculated for each word.
To convert the polarity score returned by TextBlob we use **pd.cut** function to perform **binning - convert continuous variable to a categorical variable.**

For our polarity score we use **'polarity'** result.

Vedar (Valence Aware Dictionary and sEntiment Reasoner) uses a sentiment lexicon that contains intensity measures for each word based on human-annotated labels, and designed with a focus on social media texts.

Its extracts **'neg' 'neu' 'pos' and 'compound' , each is part of the polarity 'score'**

For our polarity score we use **'compound'** result.

To convert the polarity score returned by TextBlob and Vader, and predict the target we use **pd.cut** function to perform **binning - convert continuous variable to a categorical variable.**

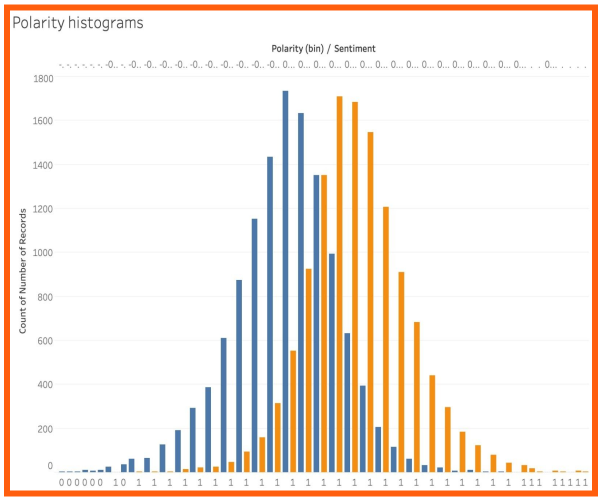
```
def _predict(self):
    self.test_df[FieldsEnum.SCORE] = self.test_df.apply(self._score, axis=1)
    self.test_df[FieldsEnum.PRED] = pd.cut(self.test_df[FieldsEnum.SCORE],
                                           bins=2,
                                           labels=[0, 1])
```

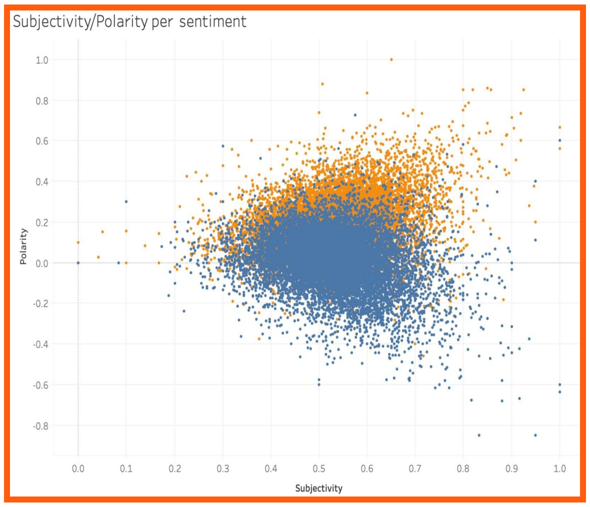
Polarity feature based - Classification models using Sentiment Scoring features

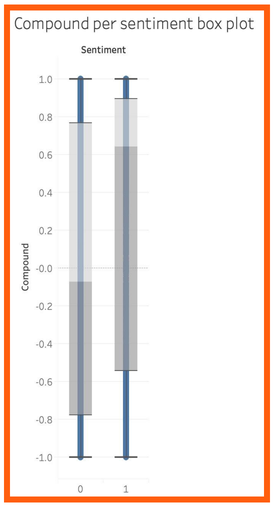
We've used all the features extracted by textBlob and Vader while adding statistical features like: 'MaxPolarity', 'AVGPolarity', 'STDPolarity', 'numPositive', 'numNegative' - counting up to **12 new features** on the dataset with no numerical features at all.

Failed attempts- We've tried to extract review 'word count' and 'sentence count' but the correlation to the sentiment target was not existing, as we thought.

Before running the prediction models, we've extracted some investing visualisations:





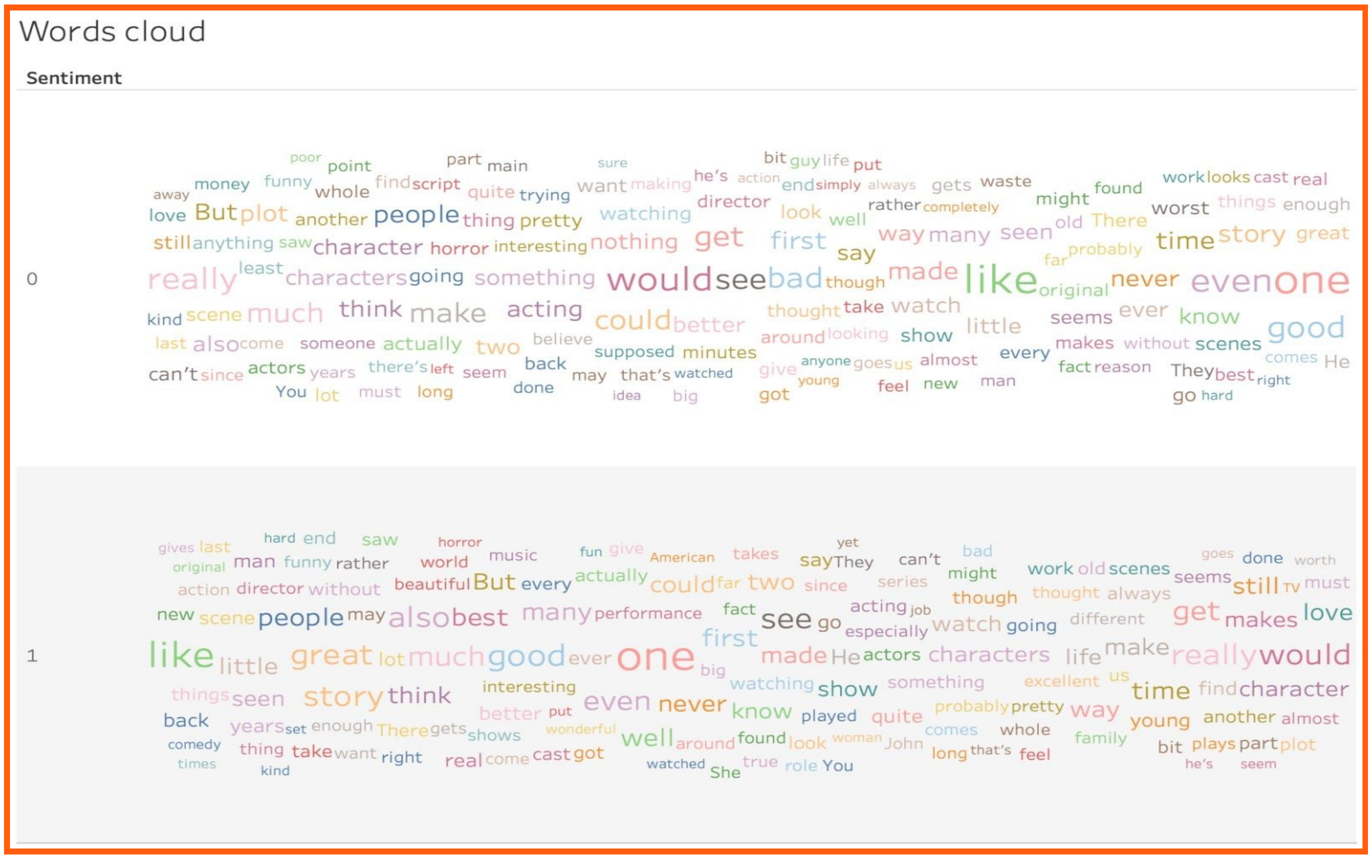


Afterwards we've used the new dataset with the new features and run the following classification models: **LogisticRegression, RandomForestClassifier, LinearSVC, SVM**

Feature Based method - TF IDF model (based on BoW)

We've used the traditional **tf-idf** based **bag-of-words (BoW)** model to extract word features and run the same classification models as before to do prediction of the sentiment target - classification models: **LogisticRegression, RandomForestClassifier, LinearSVC, SVM , MultinomialNB**

```
class FeatureBasedClass(Base):
   def __init__(self, clf, name):
       super().__init__(name)
       self.pipeline = Pipeline([('vect', CountVectorizer()), ('tfidf',
                                 TfidfTransformer()), ('clf', clf)])
   def _generate_model(self):
       # type: () -> None
       self.model = self.pipeline.fit(self.train_df[FieldsEnum.REVIEW],
                                      self.train_df[FieldsEnum.SENTIMENT])
   def _predict(self):
       # type: () -> None
       self.test_df[FieldsEnum.PRED]=
                    pd.Series(self.model.predict(self.test_df[FieldsEnum.REVIEW]))
       self.test_df.fillna(0, inplace=True)
```



Embedding based methods

FastText (Facebook)

FastText is a **CPU-based library for text representation and classification**, was released by the Facebook AI Research (FAIR) team in 2016. Its underlying neural network learns representations, or **embeddings** that **consider similarities between words**. FastText considers **subwords using a collection of n-grams** and **able to break down long words into subwords** that might also appear in other long words, **giving it better context.**

```python
class FastTextSentiment(Base):

   def __init__(self):
       super().__init__('FastText')

   def _prepare_data(self, train_labeled_data):
       data = self.train_df[[FieldsEnum.SENTIMENT, FieldsEnum.REVIEW]]

       data[FieldsEnum.SENTIMENT_LABEL] = ['__label__' + str(s) for s in
                                           data[FieldsEnum.SENTIMENT]]

       data[FieldsEnum.REVIEW] = data[FieldsEnum.REVIEW].replace('\n', ' ',
                                       regex=True).replace('\t', ' ', regex=True)

       data[[FieldsEnum.SENTIMENT_LABEL, FieldsEnum.REVIEW]].to_csv(train_labeled_data,
       index=False, sep=' ',header=False, quoting=csv.QUOTE_NONE, quotechar="",
       escapechar=" ")

   def _generate_model(self):
       labeled_trained_data = os.path.join(__path_to_base__, 'dataset_with_labels.txt')
       self._prepare_data(labeled_trained_data)
       self.model = fasttext.train_supervised(input=labeled_trained_data, **hyper_params)

   def _score(self, text):
       labels, probabilities = self.model.predict(text[FieldsEnum.REVIEW])
       pred = int(labels[0][-1])
       return pred

   def _predict(self):
       self.test_df[FieldsEnum.PRED] = self.test_df.apply(self._score, axis=1)
```

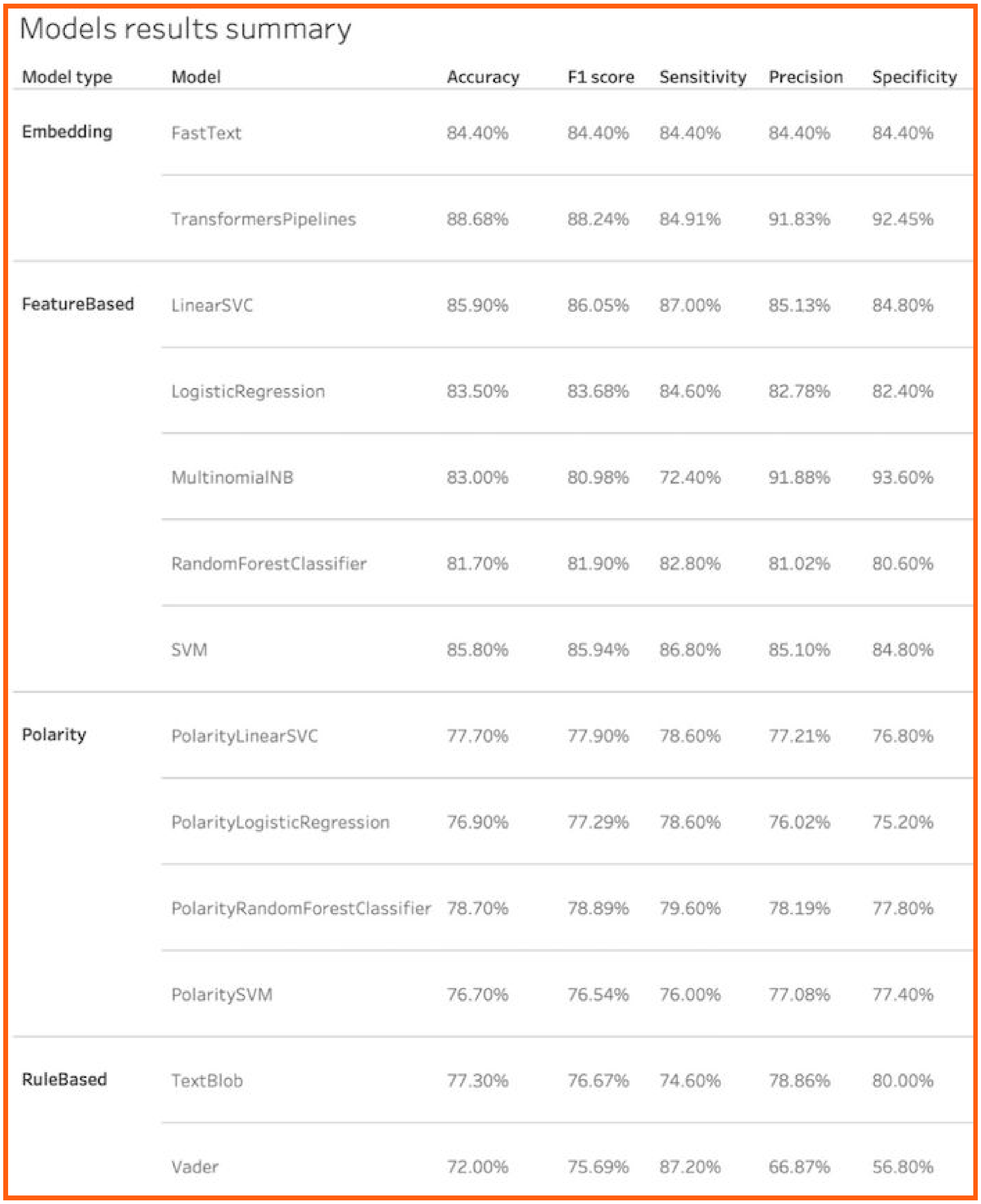
Transformers

Transformers (AKA pytorch-transformers and pytorch-pretrained-bert) provides **state-of-the-art general-purpose architectures** (BERT, GPT-2, RoBERTa, XLM, DistilBert, XLNet, CTRL etc.) for **Natural Language Understanding (NLU) and Natural Language Generation (NLG)** with over 32+ pretrained models in 100+ languages and deep interoperability between TensorFlow 2.0 and PyTorch.
https://github.com/huggingface/transformers#quick-tour

We used **transformers Pipelines** - a wrapper around tokenizer and models to use finetuned models

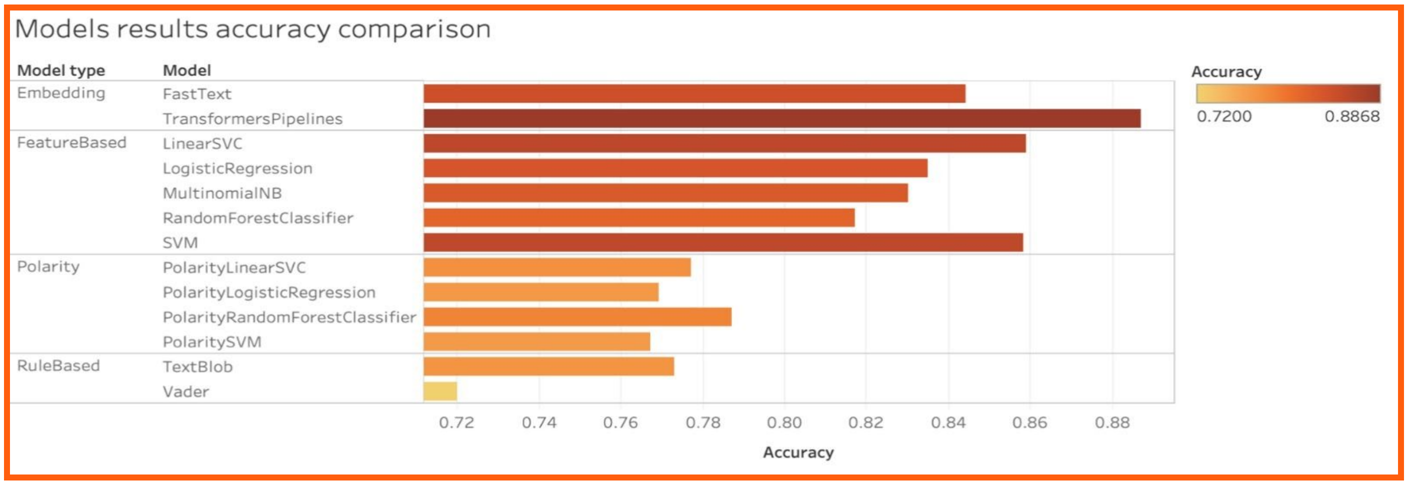
```
class TransformersPipelines:

   def _init_(self):
       self.name = 'TransformersPipelines'
       self.data = pd.read_csv(sys.argv[1])
       self.output_csv = os.path.join(os.path.dirname(sys.argv[1]), 'results',
                                       '{}.csv'.format(self.name))
       # this is the transformers model
       self.model = pipeline('sentiment-analysis')

   def _dump_results(self):
       self.data[[FieldsEnum.SENTIMENT, FieldsEnum.PRED]].to_csv(self.output_csv,
index=False)

   def _score(self, row):
       if row.name % 100 == 0:
           print(row.name)
       return 0 if self.model(row[FieldsEnum.REVIEW])[0]['label'] == 'NEGATIVE'
else 1

   def _predict(self):
       self.data[FieldsEnum.PRED] = self.data.apply(self._score, axis=1)

   def run(self):
       self._predict()
       self._dump_results()
```

Results



Models results summary

Model type	Model	Accuracy	F1 score	Sensitivity	Precision	Specificity
Embedding	FastText	84.40%	84.40%	84.40%	84.40%	84.40%
	TransformersPipelines	88.68%	88.24%	84.91%	91.83%	92.45%
FeatureBased	LinearSVC	85.90%	86.05%	87.00%	85.13%	84.80%
	LogisticRegression	83.50%	83.68%	84.60%	82.78%	82.40%
	MultinomialNB	83.00%	80.98%	72.40%	91.88%	93.60%
	RandomForestClassifier	81.70%	81.90%	82.80%	81.02%	80.60%
	SVM	85.80%	85.94%	86.80%	85.10%	84.80%
Polarity	PolarityLinearSVC	77.70%	77.90%	78.60%	77.21%	76.80%
	PolarityLogisticRegression	76.90%	77.29%	78.60%	76.02%	75.20%
	PolarityRandomForestClassifier	78.70%	78.89%	79.60%	78.19%	77.80%
	PolaritySVM	76.70%	76.54%	76.00%	77.08%	77.40%
RuleBased	TextBlob	77.30%	76.67%	74.60%	78.86%	80.00%
	Vader	72.00%	75.69%	87.20%	66.87%	56.80%



Conclusions

When using simple static models, the predictions results tend to be a bit better than a dummy model, where it usually has ~50% accuracy.

We were surprised to see that statistics models, based on features extracted from the polarity methods, had pretty much the same results as the rule-based. During the EDA step, we've noticed that there is a correlation between the polarity scores and the sentiments. The plots we extracted made us think that a sophisticated model, based on these features, might have better results. Perhaps the models are lacked features which might help better explain the data.

When using the common models for NLP/sentiment analysis (TFIDF+statistics models) the results get better than the previous two and almost get to 86% accuracy. It was interesting to see that there was a variety in the metrics. Some models had bad sensitivity grades, however compensated with better precision results. When focusing on the metric that more relevant, one can choose one model instead of the other.

Eventually, we used embedding type of models and got even better accuracy results. However, it came with a cost - longer processing time. Another disadvantage of these types of models, is the lack of capability to understand the mechanism behind.

Moreover we've tried running document embedding with RNN and ELMO/BRET word embedding, but without the proper HW for it, we were unable to get useful results, hence omitted from this paper.

Bottom line, it can be concluded from the results that more complicated model might do better than simpler ones, however the time and the tools needed might be a barrier for use. An over simplified models won't perform well in most cases. Depending on the accuracy needed, common model might be the best solution and the relevant metric can be the fine tuning for choosing the ideal model.