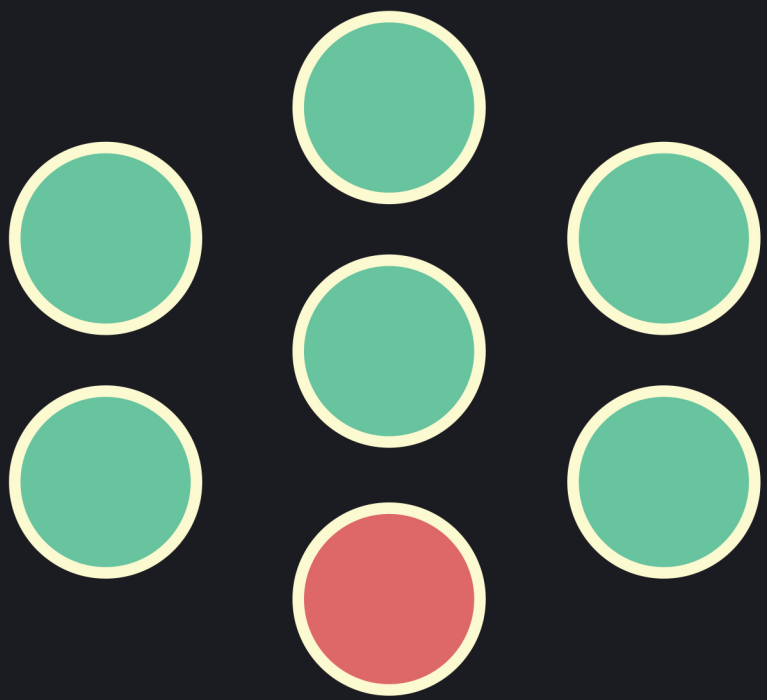
Venelin Valkov's

# Hands-On Machine Learning from Scratch



# Hands-On Machine Learning from Scratch

## Develop a Deeper Understanding of Machine Learning Models by Implementing Them from Scratch in Python

Venelin Valkov

This book is for sale at http://leanpub.com/hmls

This version was published on 2019-06-29


This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

# Tweet This Book!

Please help Venelin Valkov by spreading the word about this book on Twitter!

The suggested hashtag for this book is #hmls.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#hmls

# Contents

# Color palette extraction with K-means clustering

Choosing a color palette for your next big mobile app (re)design can be a daunting task, especially when you don't know what the heck you're doing. How can you make it easier (asking for a friend)?

One approach is to head over to a place where the *PROs* share their work. Pages like Dribbble[1], uplabs[2] and Behance[3] have the goods.

After finding mockups you like, you might want to extract colors from them and use those. This might require opening specialized software, manually picking color with some tool(s) and other over-the-counter hacks. Let's use Machine Learning to make your life easier.

**Complete source code notebook**[4]

## Unsupervised Learning

Up until now we only looked at models that require training data in the form of features and labels. In other words, for each example, we need to have the correct answer, too.

Usually, such training data is hard to obtain and requires many hours of manual work done by humans (yes, we're already serving "The Terminators"). Can we skip all that?

Yes, at least for some problems we can use example data without knowing the correct answer for it. One such problem is *clustering*.

### What is clustering?

Given some vector of data points $X$, clustering methods allow you to put each point in a group. In other words, you can categorize a set of entities, based on their properties, automatically.

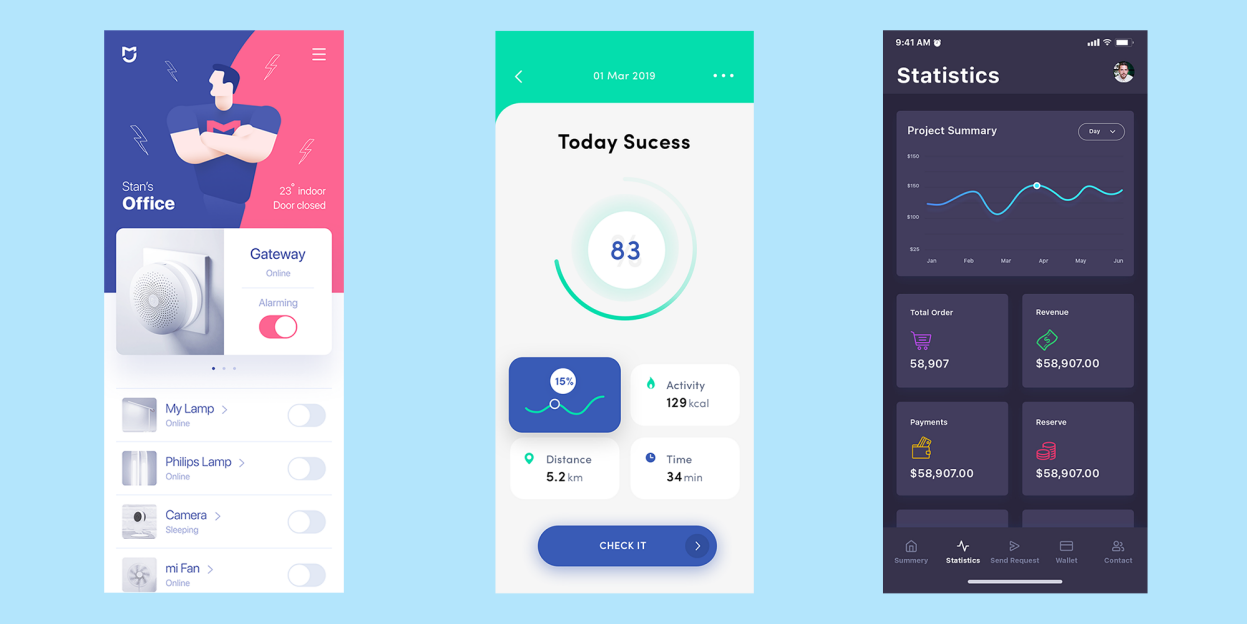
Yes, that is very useful in practice. Usually, you run the algorithm on a bunch of data points and specify how much groups you want. For example, your inbox contains two main groups of e-mail: spam and not-spam (were you waiting for something else?). You can let the clustering algorithm create two groups from your emails and use your beautiful brain to classify which is which.

More applications of clustering algorithms:

[1] https://dribbble.com/
[2] https://www.uplabs.com/
[3] https://www.behance.net/
[4] https://colab.research.google.com/drive/1_p1nptDfvJhcSvprmi1WtoIugsCI40Xa

- *Customer segmentation* - find groups of users that spend/behave the same way
- *Fraudulent transactions* - find bank transactions that belong to different clusters and identify them as fraudulent
- *Document analysis* - group similar documents

# The Data




*source: various authors on https://www.uplabs.com/*

This time, our data doesn't come from some predefined or well-known dataset. Since Unsupervised learning does not require labeled data, the Internet can be your oyster.

Here, we'll use 3 mobile UI designs from various authors. Our model will run on each shot and try to extract the color palette for each.

# What is K-Means Clustering?

K-Means Clustering is defined by Wikipedia[5] as:

> k-means clustering is a method of vector quantization, originally from signal processing, that is popular for cluster analysis in data mining. k-means clustering aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. This results in a partitioning of the data space into Voronoi cells.

[5] https://en.wikipedia.org/wiki/K-means_clustering

Wikipedia also tells us that solving K-Means clustering is difficult (in fact, NP-hard[6]) but we can find local optimum solutions using some heuristics.

But how do *K-Means Clustering* works?

Let's say you have some vector $X$ that contains $n$ data points. Running our algorithm consists of the following steps:

1. Take random $k$ points (called *centroids*) from $X$
2. Assign every point to the closest centroid. The newly formed bunch of points is called *cluster*.
3. For each cluster, find new centroid by calculating a new center from the points
4. Repeat steps 2-3 until centroids stop changing

Let's see how can we use it to extract color palettes from mobile UI screenshots.

# Data Preprocessing

Given our data is stored in raw pixels (called images), we need a way to convert it to points that our clustering algorithm can use.

Let's first define two classes that represent a point and cluster:

```
class Point:

  def __init__(self, coordinates):
    self.coordinates = coordinates
```

Our `Point` is just a holder to the coordinates for each dimension in our space.

```
class Cluster:

  def __init__(self, center, points):
    self.center = center
    self.points = points
```

The `Cluster` is defined by its center and all other points it contains.

Given a path to image file we can create the points as follows:

[6] https://en.wikipedia.org/wiki/NP-hardness

```
def get_points(image_path):
  img = Image.open(image_path)
  img.thumbnail((200, 400))
  img = img.convert("RGB")
  w, h = img.size

  points = []
  for count, color in img.getcolors(w * h):
    for _ in range(count):
      points.append(Point(color))

  return points
```

A couple of things are happening here:

- load the image into memory
- resize it to smaller image (mobile UX requires big elements on the screen, so we aren't losing much color information)
- drop the alpha (transparency) information

Note that we're creating a `Point` for each pixel in our image.

Alright! You can now extract points from an image. But how can we calculate the distance between points in our clusters?

## Distance function

Similar to the cost function in our supervised algorithm examples, we need a function that tells us how well we're doing. The objective of our algorithm is to minimize the distance between the points in each centroid.

One of the simplest distance functions we can use is the Euclidean distance, defined by:

$$d(p, q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

where $p$ and $q$ are two points from our space.

Note that while Euclidean distance is simple to implement it might not be the best way to calculate the color difference[7].

Here is the Python implementation:

[7] https://en.wikipedia.org/wiki/Color_difference

```
def euclidean(p, q):
  n_dim = len(p.coordinates)
  return sqrt(sum([
      (p.coordinates[i] - q.coordinates[i]) ** 2 for i in range(n_dim)
  ]))
```

## Implementing K-Means clustering

Now that you have all pieces of the puzzle you can implement the K-Means clustering algorithm. Let's start with the method that finds the center for a set of points:

```
def calculate_center(self, points):
  n_dim = len(points[0].coordinates)
  vals = [0.0 for i in range(n_dim)]
  for p in points:
    for i in range(n_dim):
      vals[i] += p.coordinates[i]
  coords = [(v / len(points)) for v in vals]
  return Point(coords)
```

To find the center of a set of points, we add the values for each dimension and divide by the number of points.

Now for finding the actual clusters:

```
def assign_points(self, clusters, points):
  plists = [[] for i in range(self.n_clusters)]

  for p in points:
    smallest_distance = float('inf')

    for i in range(self.n_clusters):
      distance = euclidean(p, clusters[i].center)
      if distance < smallest_distance:
        smallest_distance = distance
        idx = i

    plists[idx].append(p)

  return plists

```

```
def fit(self, points):
  clusters = [Cluster(center=p, points=[p]) for p in random.sample(points, self.n_cl\
usters)]

  while True:

    plists = self.assign_points(clusters, points)

    diff = 0

    for i in range(self.n_clusters):
      if not plists[i]:
        continue
      old = clusters[i]
      center = self.calculate_center(plists[i])
      new = Cluster(center, plists[i])
      clusters[i] = new
      diff = max(diff, euclidean(old.center, new.center))

    if diff < self.min_diff:
      break

  return clusters
```

The implementation follows the description of the algorithm given above. Note that we exit the training loop when the color difference is lower than the one set by us.

# Evaluation

Now that you have an implementation of K-Means clustering you can use it on the UI screenshots. We need a little more glue code to make it easier to extract color palettes:

```
def rgb_to_hex(rgb):
  return '#%s' % ''.join(('%02x' % p for p in rgb))

def get_colors(filename, n_colors=3):
  points = get_points(filename)
  clusters = KMeans(n_clusters=n_colors).fit(points)
  clusters.sort(key=lambda c: len(c.points), reverse = True)
  rgbs = [map(int, c.center.coordinates) for c in clusters]
  return list(map(rgb_to_hex, rgbs))
```

The `get_colors` function takes a path to an image file and the number of colors you want to extract from the image. We sort the clusters obtained from our algorithm based on the points in each (in descending order). Finally, we convert the RGB colors to hexadecimal values.

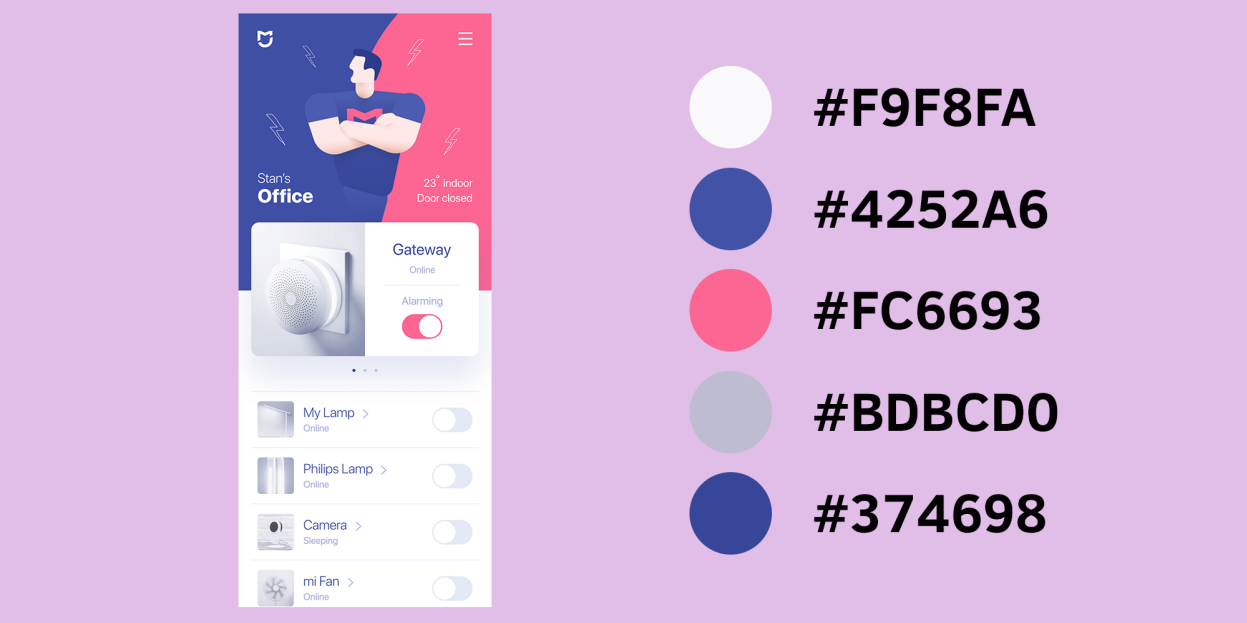
Let's extract the palette for the first UI screenshot from our data:

```
colors = get_colors('ui1.png', n_colors=5)
", ".join(colors)
```

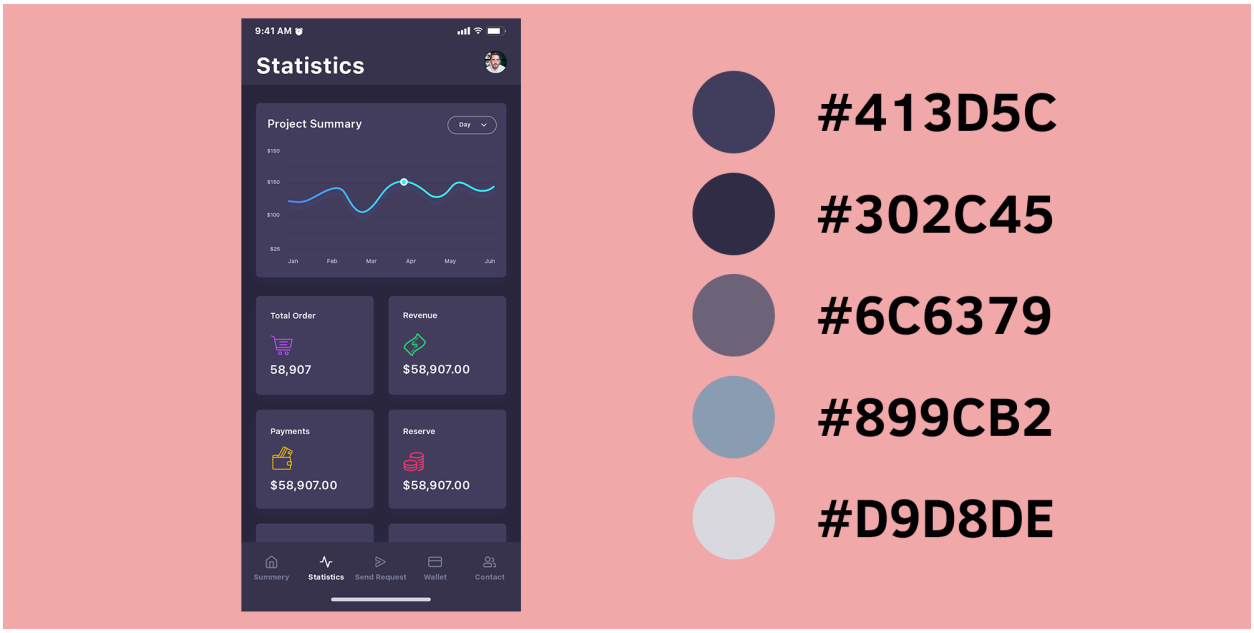
and here are the hexadecimal color values:

```
#f9f8fa, #4252a6, #fc6693, #bdbcd0, #374698
```

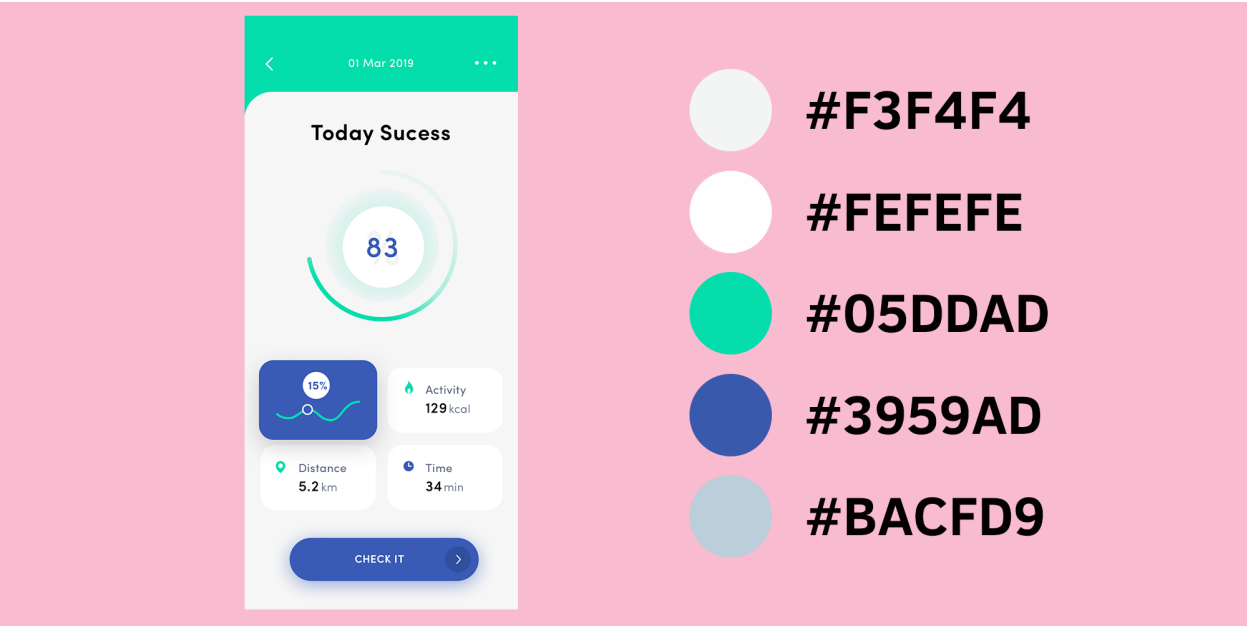
And for a visual representation of our results:




Running the clustering on the next two images, we obtain the following:




The last screenshot:




Well, that looks cool, right? Go on try it on your own images!

**Complete source code notebook on Google Colaboratory**[8]

[8] https://colab.research.google.com/drive/1_p1nptDfvJhcSvprmi1WtoIugsCI40Xa

# Conclusion

Congratulations, you just implemented your first unsupervised algorithm from scratch! It also appears that it obtains good results when trying to extract a color palette from images.

Next up, we're going to do sentiment analysis on short phrases and learn a bit more how we can handle text data.

# Movie review sentiment analysis with Naive Bayes

> TL;DR Build Naive Bayes text classification model using Python from Scratch. Use the model to classify IMDB movie reviews as positive or negative.

Textual data dominates our world from the tweets you read to the timeless writings of Seneca. And while we're consuming images (looking at you Instagram) and videos at increasing rates, you still read Google search results multiple times per daily.

One frequently recurring problem with text data is Sentiment analysis[9] (classification). Imagine you're a big sugar + water beverage company. You want to know what people think of your products. You'll search for texts with some tags, logos or names. You can then use Sentiment analysis to figure out if the opinions are positive or negative. Of course, you'll send the negative ones to your highly underpaid support center in India to sort things out.

Here, we'll build a generic text classifier that puts movie review texts into one of two categories - negative or positive sentiment. We're going to have a brief look at the Bayes theorem and relax its requirements using the Naive assumption.

Complete source code in Google Colaboratory Notebook[10]

## Dealing with Text

Computers don't understand text data, though they do well with numbers. Natural Language Processing (NLP)[11] offers a set of approaches to solve text-related problems and represent text as numbers. While NLP is a vast field, we'll use some simple preprocessing techniques and Bag of Words[12] model.

### The Data

Our data comes from a Kaggle challenge - "Bag of Words Meets Bags of Popcorn"[13].

We have 25,000 movie reviews from IMDB labeled as positive or negative. You might know that IMDB ratings are in the 0-10 range. An additional preprocessing step, done by the dataset authors, converts the rating to binary sentiment (<5 - negative ). Of course, a single movie can have multiple reviews, but no more than 30.

[9] https://en.wikipedia.org/wiki/Sentiment_analysis
[10] https://colab.research.google.com/drive/1OPQDDJTKy0b40pziZWSsoBCmQV6HyXsm
[11] https://en.wikipedia.org/wiki/Natural_language_processing
[12] https://en.wikipedia.org/wiki/Bag-of-words_model
[13] https://www.kaggle.com/c/word2vec-nlp-tutorial

## Reading the reviews

Let's load the training and test data in Pandas data frames:

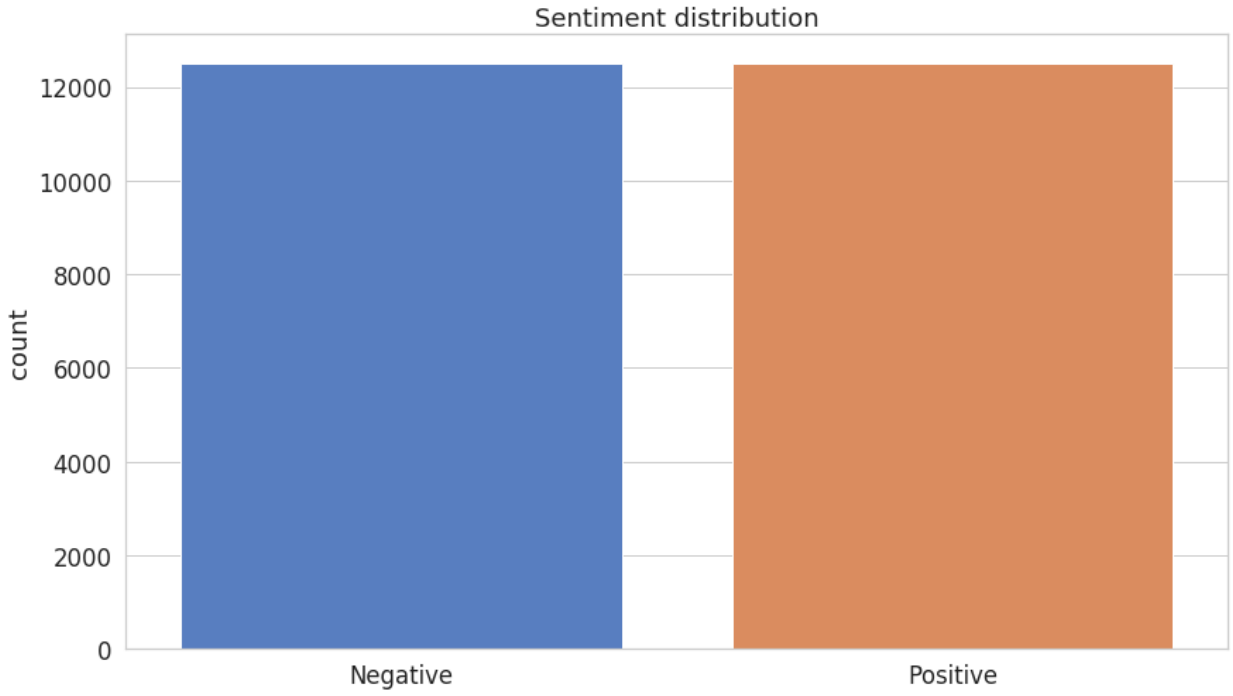
```
train = pd.read_csv("imdb_review_train.tsv", delimiter="\t")
test = pd.read_csv("imdb_review_test.tsv", delimiter="\t")
```

## Exploration

Let's get a feel for our data. Here are the first 5 rows of the training data:

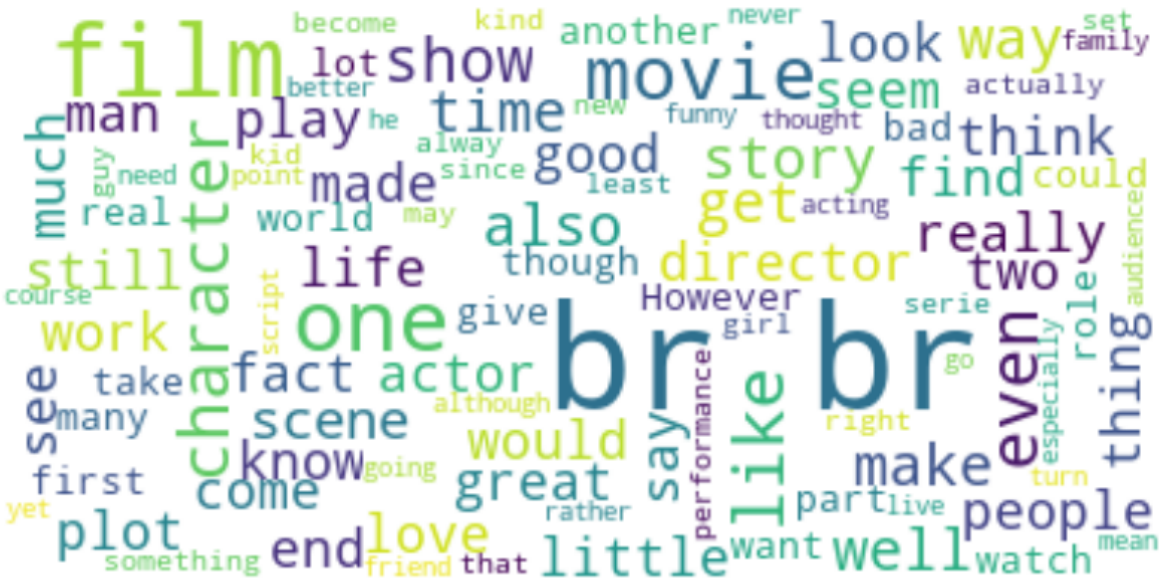
| id | sentiment | review |
|---|---|---|
| 5814_8 | 1 | With all this stuff going down at the moment w... |
| 2381_9 | 1 | The Classic War of the Worlds" by Timothy Hi... |
| 7759_3 | 0 | The film starts with a manager (Nicholas Bell)... |
| 3630_4 | 0 | It must be assumed that those who praised this... |
| 9495_8 | 1 | Superbly trashy and wondrously unpretentious 8... |

We're going to focus on the sentiment and review columns. The id column is combining the movie id with a unique number of a review. This might be a piece of important information in real-world scenarios, but we're going to keep it simple.




Both positive and negative sentiments have an equal presence. No need for additional gimmicks to fix that!

Here are the most common words in the training dataset reviews:




Hmm, that **br** looks weird, right?

## Cleaning

Real-world text data is really messy. It can contain excessive punctuation, HTML tags (including that br), multiple spaces, etc. We'll try to remove/normalize most of it.

Most of the cleaning we'll do using Regular Expressions[14], but we'll use two libraries to handle HTML tags (surprisingly hard to remove) and removing common (stop) words:

```
1  def clean(self, text):
2      no_html = BeautifulSoup(text).get_text()
3      clean = re.sub("[^a-z\s]+", " ", no_html, flags=re.IGNORECASE)
4      return re.sub("(\s+)", " ", clean)
```

First, we use BeautifulSoup[15] to remove HTML tags from our text. Second, we remove anything that is not a letter or space (note the ignoring of uppercase characters). Finally, we replace excessive spacing with a single one.

[14] https://en.wikipedia.org/wiki/Regular_expression
[15] https://www.crummy.com/software/BeautifulSoup/bs4/doc/

## Tokenization

Now that our reviews are "clean", we can further prepare them for our Bag of Words model. Let's them to lowercase letters and split them into individual words. This process is known as tokenization[16]:

```
def tokenize(self, text):
    clean = self.clean(text).lower()
    stopwords_en = stopwords.words("english")
    return [w for w in re.split("\W+", clean) if not w in stopwords_en]
```

The last step of our pre-processing is to remove stop words[17] using those defined in the NLTK[18] library. Stop words are usually frequently occurring words that might not significantly affect the meaning of the text. Some examples in English are: "is", "the", "and".

An additional benefit of removing stop words is speeding up our models since we're removing the amount of train/test data. However, in real-world scenarios, you should think about whether removing stop words can be justified.

We'll place our clean and tokenize function in a class called *Tokenizer*.

# Naive Bayes

*Naive Bayes* models are probabilistic classifiers that use the Bayes theorem[19] and make a strong assumption that the features of the data are independent. For our case, this means that each word is independent of others.

Intuitively, this might sound like a dumb idea. You know that (even from reading) the prev word(s) influence the current and next ones. However, the assumption simplifies the math and works really well in practice[20]!

The Bayes theorem is defined as:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

where $A$ and $B$ are some events and $P(.)$ is probability.

This equation gives us the conditional probability of event $A$ occurring given $B$ has happened. In order to find this, we need to calculate the probability of $B$ happening given $A$ has happened and

[16] https://en.wikipedia.org/wiki/Lexical_analysis#Tokenization
[17] https://en.wikipedia.org/wiki/Stop_words
[18] https://www.nltk.org/
[19] https://en.wikipedia.org/wiki/Bayes%27_theorem
[20] https://www.cc.gatech.edu/~isbell/reading/papers/Rish.pdf

multiply that by the probability of $A$ (known as *Prior*) happening. All of this is divided by the probability of $B$ happening on its own.

The naive assumption allows us to reformulate the Bayes theorem for our example as:

$$P(Sentiment|w_1,\ldots,w_n) = \frac{P(Sentiment)\prod_{i=1}^{n} P(w_i|Sentiment)}{P(w_1,\ldots,w_n)}$$

We don't really care about probabilities. We only want to know whether a given text has a positive or negative sentiment. We can skip the denominator entirely since it just scales the numerator:

$$P(Sentiment|w_1,\ldots,w_n) \propto P(Sentiment)\prod_{i=1}^{n} P(w_i|Sentiment)$$

To choose the sentiment, we'll compare the scores for each sentiment and pick the one with a higher score.

# Implementing Multinomial Naive Bayes

As you might've guessed by now, we're classifying text into one of two groups/categories - positive and negative sentiment.

Multinomial Naive Bayes allows us to represent the features of the model as frequencies of their occurrences (how often some word is present in our review). In other words, it tells us that the probability distributions we're using are multinomial[21].

## Note on numerical stability

Our model relies on multiplying many probabilities. Those might become increasingly small and our computer might round them down to zero. To combat this, we're going to use log probability by taking log of each side in our equation:

$$\log P(Sentiment|w_1,\ldots,w_n) = \log P(Sentiment) + \log\prod_{i=1}^{n} P(w_i|Sentiment)$$

Note that we can still use the highest score of our model to predict the sentiment since log is monotonic[22].

[21] https://en.wikipedia.org/wiki/Multinomial_distribution
[22] https://en.wikipedia.org/wiki/Monotonic_function

## Building our model

Finally, time to implement our model in Python. Let's start by defining some variables and group the data by class, so our training code is a bit tidier:

```
def fit(self, X, y):
    self.n_class_items = {}
    self.log_class_priors = {}
    self.word_counts = {}
    self.vocab = set()

    n = len(X)
    grouped_data = self.group_by_class(X, y)
    ...
```

We're going to implement a generic text classifier that doesn't assume the number of classes. You can use it for news category prediction, sentiment analysis, email spam detection, etc.

For each class, we'll find the number of examples in it and the log probability (prior). We'll also record the number of occurrences of each word and create a vocabulary - set of all words we've seen in the training data:

```
    for c, data in grouped_data.items():
      self.n_class_items[c] = len(data)
      self.log_class_priors[c] = math.log(self.n_class_items[c] / n)
      self.word_counts[c] = defaultdict(lambda: 0)

      for text in data:
        counts = Counter(self.tokenizer.tokenize(text))
        for word, count in counts.items():
            if word not in self.vocab:
                self.vocab.add(word)

            self.word_counts[c][word] += count
```

Note that we use our *Tokenizer* and the built-in class Counter[23] to convert a review to a bag of words.

In case you're interested, here's how `group_by_class` works:

[23] https://docs.python.org/3/library/collections.html#collections.Counter

```
def group_by_class(self, X, y):
  data = dict()
  for c in self.classes:
    data[c] = X[np.where(y == c)]
  return data
```

## Making predictions

In order to predict sentiment from text data, we'll use our class priors and vocabulary:

```
def predict(self, X):
    result = []
    for text in X:

      class_scores = {c: self.log_class_priors[c] for c in self.classes}
      words = set(self.tokenizer.tokenize(text))

      for word in words:
          if word not in self.vocab: continue

          for c in self.classes:

            log_w_given_c = self.laplace_smoothing(word, c)
            class_scores[c] += log_w_given_c

      result.append(max(class_scores, key=class_scores.get))

    return result
```

For each review, we use the log priors for positive and negative sentiment and tokenize the text. For each word, we check if it is in the vocabulary and skip it if it is not. Then we calculate the log probability of this word for each class. We add the class scores and pick the class with a max score as our prediction.

### Laplace smoothing

Note that $log(0)$ is *undefined* (and no, we're not using JavaScript here). It is entirely possible for a word in our vocabulary to be present in one class but not another - the probability of finding this word in that class will be 0! We can use Laplace smoothing[24] to fix this problem. We'll simply add 1 to the numerator but also add the size of our vocabulary to the denominator:

[24] https://en.wikipedia.org/wiki/Additive_smoothing

```
def laplace_smoothing(self, word, text_class):
  num = self.word_counts[text_class][word] + 1
  denom = self.n_class_items[text_class] + len(self.vocab)
  return math.log(num / denom)
```

# Predicting sentiment

Now that your model can be trained and make predictions, let's use it to predict sentiment from movie reviews.

## Data preparation

As discussed previously, we'll use only the review and sentiment columns from the training data. Let split it for training and testing:

```
X = train['review'].values
y = train['sentiment'].values

X_train, X_test, y_train, y_test = train_test_split(
  X, y,
  test_size=0.2,
  random_state=RANDOM_SEED
)
```

## Evaluation

We'll pack our `fit` and `predict` functions into a class called `MultinomialNaiveBayes`. Let's use it:

```
MNB = MultinomialNaiveBayes(
    classes=np.unique(y),
    tokenizer=Tokenizer()
).fit(X_train, y_train)
```

Our classifier takes a list of possible classes and a Tokenizer as parameters. Also, the API is quite nice (thanks scikit-learn!)

```
y_hat = MNB.predict(X_test)
accuracy_score(y_test, y_hat)
```

```
0.8556
```

This looks nice we got an accuracy of ~86% on the test set.

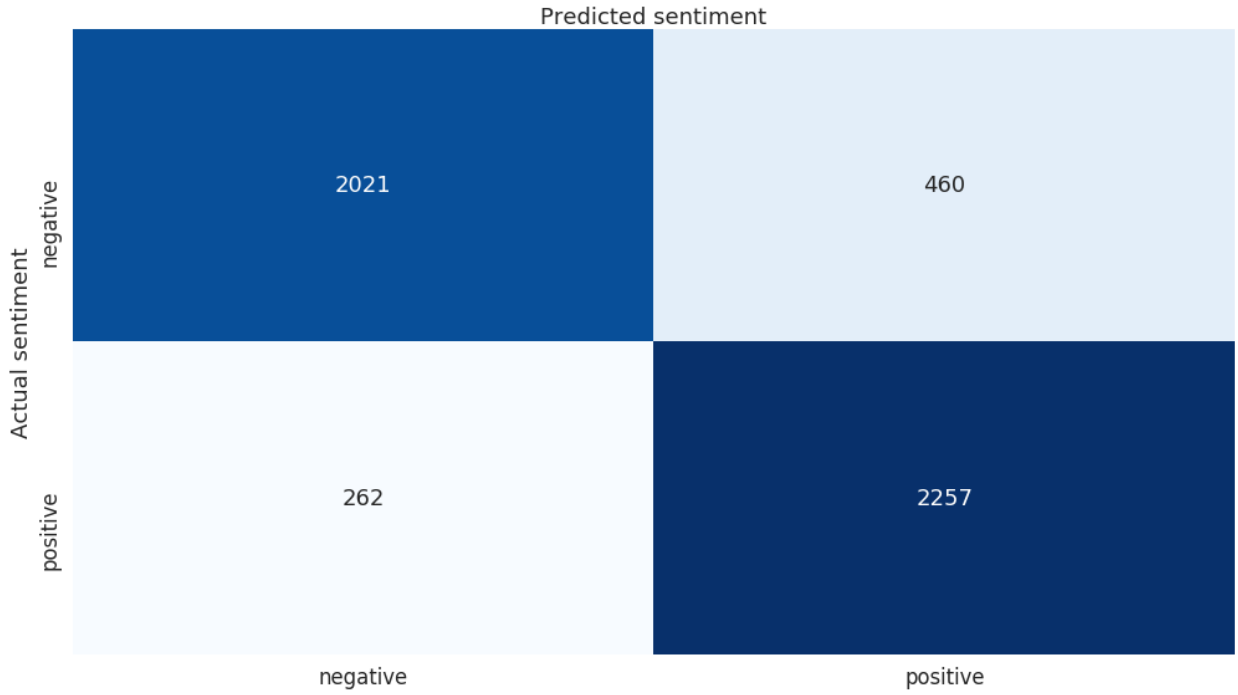
Here is the classification report:

```
              precision    recall  f1-score   support

           0       0.89      0.81      0.85      2481
           1       0.83      0.90      0.86      2519

    accuracy                           0.86      5000
   macro avg       0.86      0.86      0.86      5000
weighted avg       0.86      0.86      0.86      5000
```

And the confusion matrix:




Overall, our classifier does pretty well for himself. Submit the predictions to Kaggle and find out what place you'll get on the leaderboard.

# Conclusion

Well done! You just built a Multinomial Naive Bayes classifier that does pretty well on sentiment prediction. You also learned about Bayes theorem, text processing, and Laplace smoothing! Will another flavor of the Naive Bayes classifier perform better?

Complete source code in Google Colaboratory Notebook[25]

Next up, you'll build a recommender system from scratch!

[25] https://colab.research.google.com/drive/1OPQDDJTKy0b40pziZWSsoBCmQV6HyXsm