Sentiment analysis using TextAnalyis.jl

This notebook explores the **Twitter US Airline Sentiment** data set from Kaggle. We will use **TextAnalysis.jl** as the primary tool for analyzing textual data.

```
using Pkg
```

```
dir = "/Users/tomkwong/Julia/HumansOfJulia-WeeklyContest/Week2-TextAnalysis.jl/tk3369"
```

```
dir = "/Users/tomkwong/Julia/HumansOfJulia-WeeklyContest/Week2-TextAnalysis.jl/tk3369"
```

```
Pkg.activate(dir)
```

```
begin
    using TextAnalysis
    using CSV
    using DataFrames
    using Pipe: @pipe
end
```

Loading data

```
cd(dir)
```

```
df =
```

14,640 rows × 15 columns (omitted printing of 10 columns)

	tweet_id	airline_sentiment	airline_sentiment_confidence	negativereason	n
	Int64	String	Float64	String?	
1	570306133677760513	neutral	1.0	*missing*	n
2	570301130888122368	positive	0.3486	*missing*	0
3	570301083672813571	neutral	0.6837	*missing*	n
4	570301031407624196	negative	1.0	Bad Flight	0
5	570300817074462722	negative	1.0	Can't Tell	1
6	570300767074181121	negative	1.0	Can't Tell	0
7	570300616901320704	positive	0.6745	*missing*	0

	tweet_id	airline_sentiment	airline_sentiment_confidence	negativereason	n
	Int64	String	Float64	String?	
8	570300248553349120	neutral	0.634	*missing*	n
9	570299953286942721	positive	0.6559	*missing*	n
10	570295459631263746	positive	1.0	*missing*	n
11	570294189143031808	neutral	0.6769	*missing*	0
12	570289724453216256	positive	1.0	*missing*	n
13	570289584061480960	positive	1.0	*missing*	n
14	570287408438120448	positive	0.6451	*missing*	n
15	570285904809598977	positive	1.0	*missing*	n
16	570282469121007616	negative	0.6842	Late Flight	0
17	570277724385734656	positive	1.0	*missing*	n
18	570276917301137409	negative	1.0	Bad Flight	1
⋮	⋮	⋮	⋮	⋮	⋮

```
df = DataFrame(CSV.File("data/Tweets.csv"))
```

Data Wrangling

We will take a look at the data and get a little more understanding about what's going on in this data set.

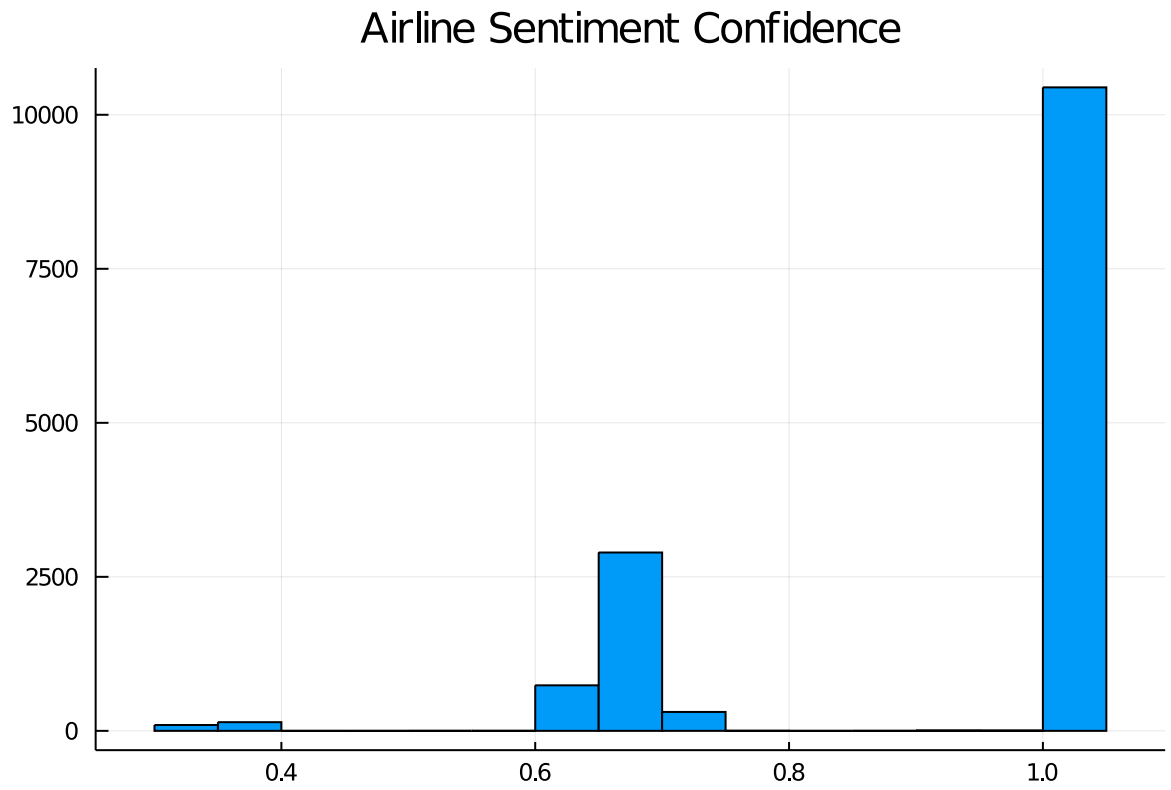
15 rows × 4 columns

	variable	eltype	nmissing	first
	Symbol	Type	Union…	Any
1	tweet_id	Int64		570306133677760513
2	airline_sentiment	String		neutral
3	airline_sentiment_confidence	Float64		1.0
4	negativereason	Union{Missing, String}	5462	Bad Flight
5	negativereason_confidence	Union{Missing, Float64}	4118	0.0
6	airline	String		Virgin America
7	airline_sentiment_gold	Union{Missing, String}	14600	negative
8	name	String		cairdin
9	negativereason_gold	Union{Missing, String}	14608	Late Flight\nFlight Attendant Complaints
10	retweet_count	Int64		0

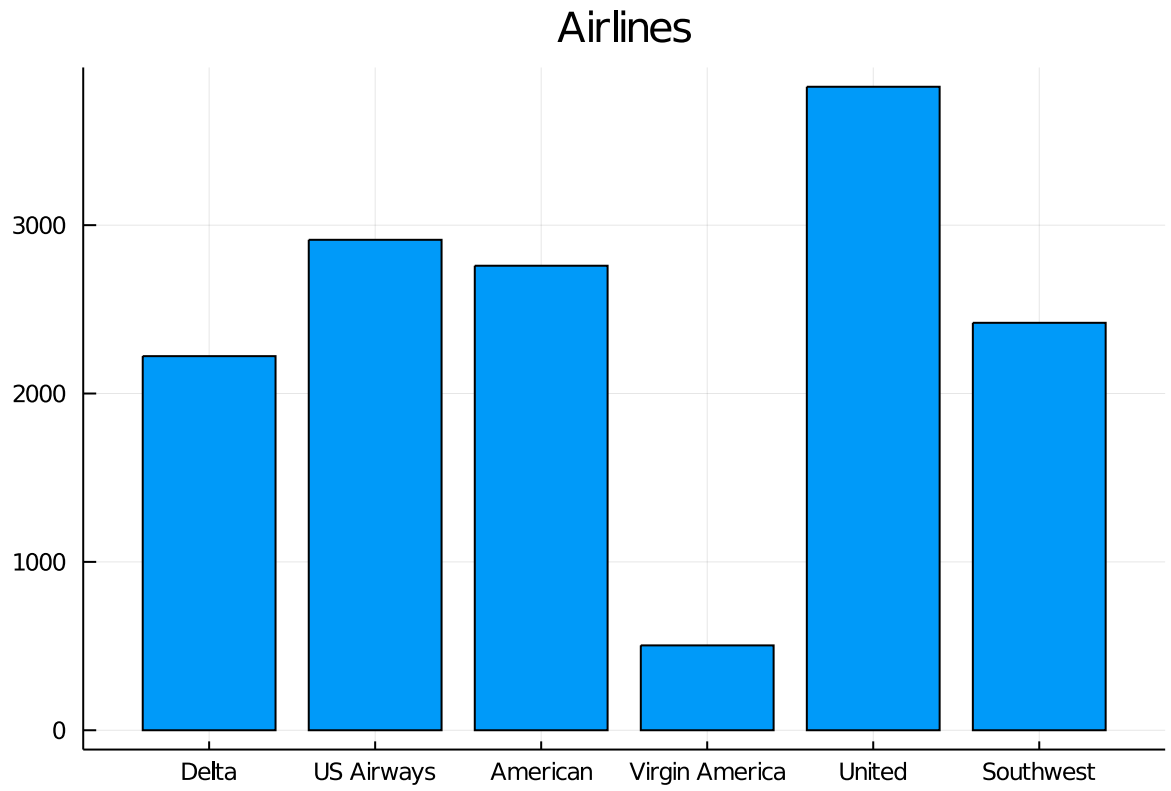
	variable Symbol	eltype Type	nmissing Union…	first Any
11	text	String		@VirginAmerica What @dhepburn said.
12	tweet_coord	Union{Missing, String}	13621	[40.74804263, -73.99295302]
13	tweet_created	String		2015-02-24 11:35:52 -0800
14	tweet_location	Union{Missing, String}	4733	Lets Play
15	user_timezone	Union{Missing, String}	4820	Eastern Time (US & Canada)

```
describe(df, :eltype, :nmissing, :first => first)
```

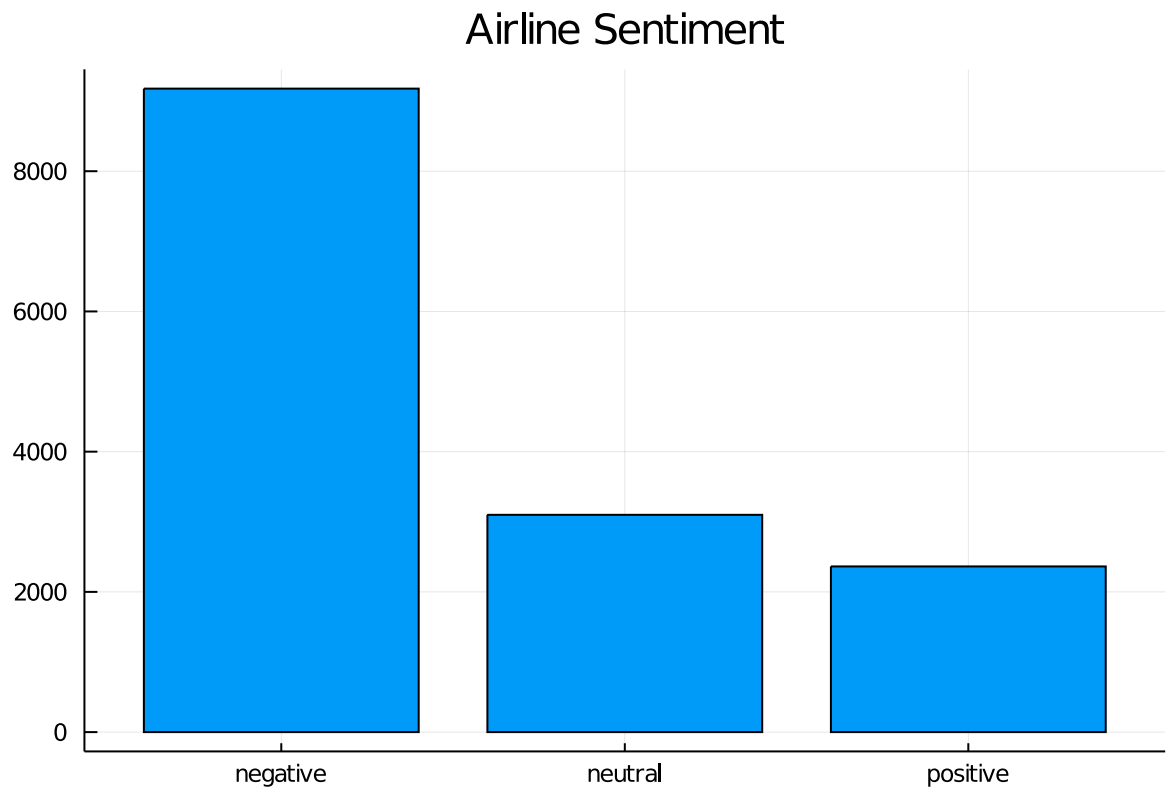
```
using Plots, StatsPlots
```



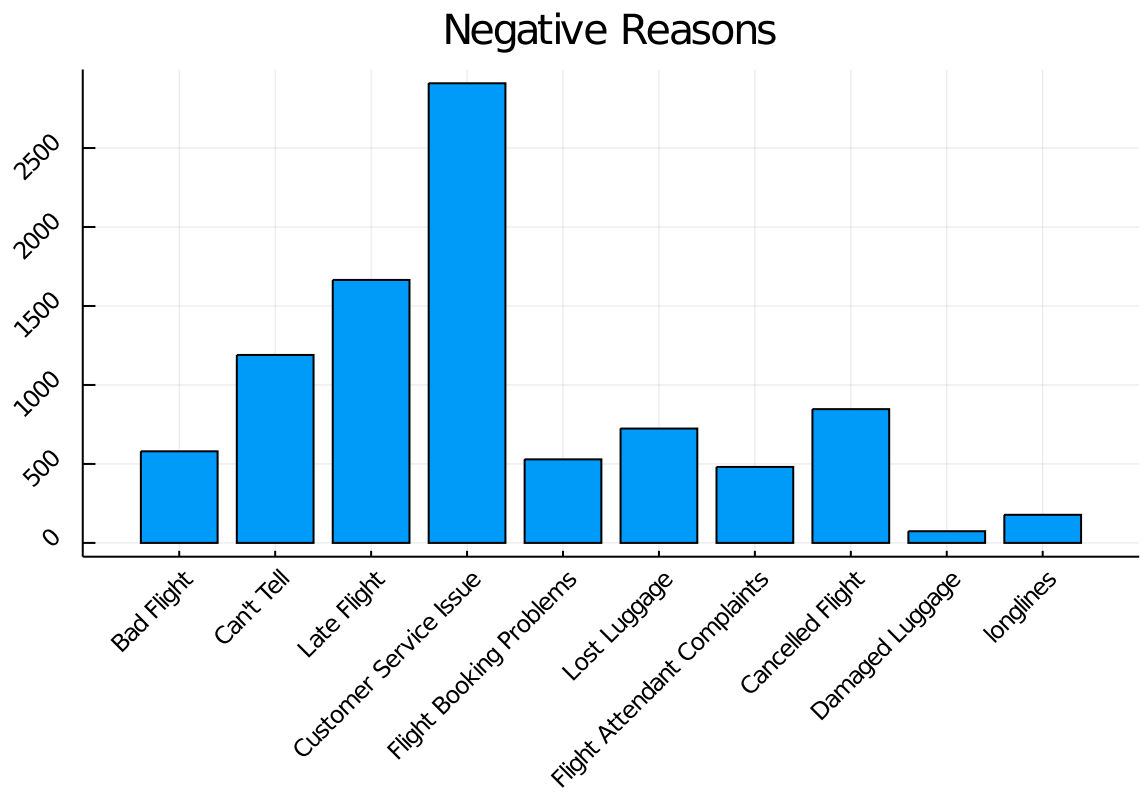

```
histogram(df.airline_sentiment_confidence;
    legend = nothing,
    title = "Airline Sentiment Confidence")
```




```
let x = combine(groupby(df, :airline), nrow)
    bar(x.airline, x.nrow; title = "Airlines", label = :none)
end
```




```
let x = combine(groupby(df, :airline_sentiment), nrow)
    bar(x.airline_sentiment, x.nrow;
        title = "Airline Sentiment",
        label = :none,
        legend = :topright)
end
```




```
let x = combine(groupby(dropmissing(df, :negativereason), :negativereason), nrow)
    bar(x.negativereason, x.nrow;
        title = "Negative Reasons", label = :none, rotation = 45)
end
```

Examining tweets

The CSV file contains over 14,000 tweets. Let's quickly examine some individual data.

Before we go further, it would be nice to display a single record in table format. We can define a `table` function that converts an indexable object into Markdown format, which can be displayed in this Pluto notebook.

```
function table(nt)
    io = IOBuffer()
    println(io, "|name|value|")
    println(io, "|---:|:----|")
    for k in keys(nt)
        println(io, "|`", k, "`|", nt[k], "|")
    end
    Markdown.parse(String(take!(io)))
end;
```

Here, we will define a variable called `row` and bind it to a slider for quick experimentation.

```
@bind row html"""<input type="range" min="1" max="$(nrow(df))" value="36"/>"""
```

```
"Current Record: 36"
```

name	value
tweet_id	570051991277342720
airline_sentiment	neutral
airline_sentiment_confidence	0.6207
negativereason	missing
negativereason_confidence	missing
airline	Virgin America
airline_sentiment_gold	missing
name	miaerolinea
negativereason_gold	missing
retweet_count	0
text	Nice RT @VirginAmerica: Vibe with the moodlight from takeoff to touchdown. #MoodlitMonday #ScienceBehindTheExperience http://t.co/Y7O0uNxTQP
tweet_coord	missing
tweet_created	2015-02-23 18:46:00 -0800
tweet_location	Worldwide
user_timezone	Caracas

```
table(df[row, :])
```

As an example, record #36 has the tweet text as:

```
Nice RT @VirginAmerica: Vibe with the moodlight from takeoff to touchdown. #MoodlitM
onday #ScienceBehindTheExperience http://t.co/Y7O0uNxTQP
```

This is a tricky one because it contins all of the followings:

- mention (`@VirginAmeria`)
- hash tag (`#MoodlitMonday` and `#ScienceBehindTheExperience`)
- URL (`http://t.co/Y7O0uNxTQP`)

Technically `RT` is a shorthand for "retweet" so perhaps it should be expanded but let's not worry about that for now.

Handling mentions and hashtags

Let's see how the default tokenizer behaves using the `ngrams` function

name	value
RT	1
moodlight	1
0	1
http://t.co/Y	1
takeoff	1
:	1
VirginAmerica	1
the	1
uNxTQP	1
7	1
#	2
with	1
Vibe	1
@	1
Nice	1
touchdown.	1
ScienceBehindTheExperience	1
0	1
MoodlitMonday	1
to	1
from	1

```
ngrams(StringDocument(df[36, :text])) |> table
```

Because mentions(`@`) and hashtags(`#`) look like punctuations, the standard tokenizer would have taken them apart from the text that follows. It would be wrong to consider someone like `@happy` as the same as `["@", "happy"]` because the tweet could have been classified with positive sentiment rather than being neutral as it is really just a reference to a user name and not literally the word "happy".

Further, the URL was broken up into pieces.

There are several ways to handle this:

1. Use `WordTokenizers.tweet_tokenize` function to tokenize the tweet. This tokenizer is aware of mentions and hashtags and it would keep them intact with the text that follows.
2. Extract mentions and hashtags from the tweet and replace them with empty string. I would think that mentions should always be considered neutral. Hashtags is a little trickier as it could be used to expressed sentiment as well. For simplicity reasons though, maybe it does not matter as much.

```
md"""
Because mentions(`@`) and hashtags(`#`) look like punctuations, the standard tokenizer would have taken them apart from the text that follows. It would be wrong to consider someone like `@happy` as the same as `["@", "happy"]` because the tweet could have
```

```
been classified with positive sentiment rather than being neutral as it is really just a reference to a user name and not literally the word "happy".

Further, the URL was broken up into pieces.

There are several ways to handle this:

1. Use `WordTokenizers.tweet_tokenize` function to tokenize the tweet. This tokenizer is aware of mentions and hashtags and it would keep them intact with the text that follows.

2. Extract mentions and hashtags from the tweet and replace them with empty string. I would think that mentions should always be considered neutral. Hashtags is a little trickier as it could be used to expressed sentiment as well. For simplicity reasons though, maybe it does not matter as much.
"""
```

For now, let's go with approach #1. Taking advice from Lyndon White, I can set the default tokenizer with the tweet tokenizer provided by `WordsTokenizers` package.

```
const WT = TextAnalysis.WordTokenizers;
```

```
WT.set_tokenizer(WT.tweet_tokenize);
```

Let's try again.

name	value
RT	1
moodlight	1
http://t.co/Y7O0uNxTQP	1
#ScienceBehindTheExperience	1
takeoff	1
.	1
:	1
the	1
@VirginAmerica	1
with	1
#MoodlitMonday	1
Vibe	1
touchdown	1
Nice	1
to	1
from	1

```
ngrams(StringDocument(df[36, :text])) |> table
```

I'll play with some preprocessing facilities.

```
using TextAnalysis: strip_punctuation, strip_stopwords, strip_punctuation
```

name	value
moodlight	1
nice	1
touchdown	1
sciencebehindtheexperi	1
moodlitmonday	1
httpcoy7o0unxtqp	1
rt	1
virginamerica	1
takeoff	1
vibe	1

```
let s = StringDocument(lowercase(df[36, :text]))
    op = 0x00
    op |= strip_punctuation
    op |= strip_stopwords
    op |= strip_html_tags
    prepare!(s, op)
    stem!(s)
    ngrams(s) |> table
end
```

Right off the bat, I can see some problems here. It seems that when I stripped punctuations, it also took the `@` and `#` signs away. The URL also became weird. Oh yeah, that's stripping punctuation means, right? :-)

Let me get rid of the `strip_punctuation` preparation step and see hat happens.

name	value
moodlight	1
@virginamerica	1
vibe	1
takeoff	1
.	1
#sciencebehindtheexperi	1
nice	1
:	1
rt	1
#moodlitmonday	1
touchdown	1
http://.co/y7o0unxtqp	1

```
let s = StringDocument(lowercase(df[36, :text]))
    op = 0x00
    op |= strip_stopwords
    op |= strip_html_tags
    prepare!(s, op)
    stem!(s)
    ngrams(s) |> table
```

```
end
```

Now, the mention and hashtag are back to normal. The URL has gotten better but something is missing. It used to be `http://t.co/Y7O0UNXTQP`. So it's missing the `t` in domain name. Apparently, the `strip_stopwords` preparation step took that away.

I'm torn. How can I make it not mess around with my mentions, hashtags, and URLs? Maybe I will take approach #2 now. I can certainly extract these data first before tokenizing the text.

This neat idea came from José Bayoán Santiago Calderón when I asked the question on Slack.

```
(  name  value , SubString{String}["@VirginAmerica"], SubString{String}["#MoodlitMond
  moodlight  1
       nice  1
  touchdown  1
         rt  1
    takeoff  1
       vibe  1
```

```
let s = df[36, :text]

    mention_regex = r"@\w+"
    hashtag_regex = r"#\w+"
    url_regex = r"http[\w:/.-]+"

    mentions = collect(x.match for x in eachmatch(mention_regex, s))
    hashtags = collect(x.match for x in eachmatch(hashtag_regex, s))
    urls = collect(x.match for x in eachmatch(url_regex, s))

    s = replace(s, mention_regex => "")
    s = replace(s, hashtag_regex => "")
    s = replace(s, url_regex => "")
    s = lowercase(s)

    sd = StringDocument(s)

    op = 0x00
    op |= strip_punctuation
    op |= strip_stopwords
    op |= strip_html_tags
    prepare!(sd, op)
    stem!(sd)
    table(ngrams(sd)), mentions, hashtags, urls
end
```

This strategy seems to work well. Let's make a copy of the data frame and start doing some analysis.

P.S. I could have modified the original data frame but I would rather not do that because it will mess up the earlier part of this Pluto notebook because the cells are reactive in Pluto.

```
function extract_mentions(s)
```

```
    mention_regex = r"@\w+"
    return collect(x.match for x in eachmatch(mention_regex, s))
end;
```

```
function extract_hashtags(s)
    hashtag_regex = r"#\w+"
    return collect(x.match for x in eachmatch(hashtag_regex, s))
end;
```

```
function extract_urls(s)
    url_regex = r"http[\w:/.]+"
    return collect(x.match for x in eachmatch(url_regex, s))
end;
```

```
function remove_extracted_text(s)
    s = replace(s, r"@\w+" => "")   # mentions
    s = replace(s, r"#\w+" => "")   # hashtags
    s = replace(s, r"http[\w:/.]+" => "")  # url's
    return s
end;
```

```
begin
    df2 = copy(df)
    transform!(df2,
        :text => ByRow(extract_mentions) => :x_mentions,
        :text => ByRow(extract_hashtags) => :x_hashtags,
        :text => ByRow(extract_urls) => :x_urls,
        :text => ByRow(remove_extracted_text) => :x_text)
end;
```

name	value
tweet_id	570051991277342720
airline_sentiment	neutral
airline_sentiment_confidence	0.6207
negativereason	missing
negativereason_confidence	missing
airline	Virgin America
airline_sentiment_gold	missing
name	miaerolinea
negativereason_gold	missing
retweet_count	0
text	Nice RT @VirginAmerica: Vibe with the moodlight from takeoff to touchdown. #MoodlitMonday #ScienceBehindTheExperience http://t.co/Y7O0uNxTQP
tweet_coord	missing
tweet_created	2015-02-23 18:46:00 -0800
tweet_location	Worldwide
user_timezone	Caracas
x_mentions	SubString{String}["@VirginAmerica"]
x_hashtags	SubString{String}["#MoodlitMonday", "#ScienceBehindTheExperience"]
x_urls	SubString{String}["http://t.co/Y7O0uNxTQP"]

x_text	Nice RT : Vibe with the moodlight from takeoff to touchdown.

```
table(df2[36, :])
```

As you can see, the mentions/hashtags/urls are extracted into separate columns in the data frame. The `x_text` field contains the cleaned version of `text`.

If we do more text processing on `x_text` column, then it looks a lot more reaonsable and without any noise.

name	value
moodlight	1
nice	1
touchdown	1
rt	1
takeoff	1
vibe	1

```
let sd = StringDocument(lowercase(df2[36, :x_text]))
    op = 0x00
    op |= strip_punctuation
    op |= strip_stopwords
    op |= strip_html_tags
    prepare!(sd, op)
    table(ngrams(sd))
end
```

Let's just define a function to do that work.

```
function tweet_string_doc(s::AbstractString)
    sd = StringDocument(lowercase(s))
    op = 0x00
    op |= strip_punctuation
    op |= strip_stopwords
    op |= strip_html_tags
    prepare!(sd, op)
    return TextAnalysis.text(sd) == "" ? missing : sd
end;
```

TextAnalysis.jl supports ngrams and so we could convert a `StringDocument` to an `NGramDocument`.

```
function tweet_ngrams(sd::StringDocument, n = 1)
    try
        return NGramDocument(ngrams(sd, n))
    catch
        return missing
    end
end;
```

Define a simple `passmissing` function to handle missing data

```
passmissing(f) = x -> ismissing(x) ? missing : f(x);
```

Now, I can create `StringDocument` and `NGramDocument` columns (both 1- and 2-grams) in the data frame.

```
begin
    transform!(df2, :x_text => ByRow(tweet_string_doc) => :x_string_doc)
    transform!(df2, :x_string_doc => ByRow(passmissing(tweet_ngrams)) => :x_ngrams)
    transform!(df2, :x_string_doc => ByRow(passmissing(x -> tweet_ngrams(x,2))) => :x_ngrams2)
    nothing
end
```

What does it look like now?

name	value
tweet_id	570051991277342720
airline_sentiment	neutral
airline_sentiment_confidence	0.6207
negativereason	missing
negativereason_confidence	missing
airline	Virgin America
airline_sentiment_gold	missing
name	miaerolinea
negativereason_gold	missing
retweet_count	0
text	Nice RT @VirginAmerica: Vibe with the moodlight from takeoff to touchdown. #MoodlitMonday #ScienceBehindTheExperience http://t.co/Y7O0uNxTQP
tweet_coord	missing
tweet_created	2015-02-23 18:46:00 -0800
tweet_location	Worldwide
user_timezone	Caracas
x_mentions	SubString{String}["@VirginAmerica"]
x_hashtags	SubString{String}["#MoodlitMonday", "#ScienceBehindTheExperience"]
x_urls	SubString{String}["http://t.co/Y7O0uNxTQP"]
x_text	Nice RT : Vibe with the moodlight from takeoff to touchdown.
x_string_doc	A TextAnalysis.StringDocument{String}
x_ngrams	A TextAnalysis.NGramDocument{AbstractString}
x_ngrams2	A TextAnalysis.NGramDocument{AbstractString}

```
table(df2[36, :])
```

Corpus analysis

Using 1-gram column, we can build a corpus and display the top 10 words.

10 rows × 2 columns

	count	word
	Int64	String
1	3901	flight
2	1070	thanks
3	1050	cancelled
4	958	service
5	842	help
6	786	time
7	751	customer
8	671	hours
9	645	2
10	643	amp

```
let
    crps = Corpus(collect(skipmissing(df2.x_ngrams)))
    update_lexicon!(crps)
    lc = lexicon(crps)

    nts = [(count = v, word = k) for (k,v) in lc]
    sort!(nts, rev = true)

    DataFrame(nts[1:10])
end
```

I can play the same trick with 2-grams.

10 rows × 2 columns

	count	word
	Int64	String
1	562	customer service
2	500	cancelled flightled
3	245	late flight
4	231	flight cancelled
5	218	cancelled flighted
6	156	late flightr
7	146	fleet fleek
8	143	cancelled flight
9	128	2 hours

	count	word
	Int64	String
10	112	flight delayed

```
let
    crps = Corpus(collect(skipmissing(df2.x_ngrams2)))
    update_lexicon!(crps)
    lc = lexicon(crps)

    nts = [(count = v, word = k) for (k,v) in lc]
    sort!(nts, rev = true)

    DataFrame(nts[1:10])
end
```

Features

Just playing with various functions in TextAnalysis.jl

```
begin
    crps = Corpus(collect(skipmissing(df2.x_string_doc)))
    update_lexicon!(crps)
end
```

```
A 14640 X 12261 DocumentTermMatrix
```

```
DocumentTermMatrix(crps)
```

```
1×12261 Array{Int64,2}:
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0  0  0
```

```
dtv(crps[1], lexicon(crps))
```

```
14640×100 Array{Int64,2}:
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  1  0  0  1  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  1  0  0
 0  0  1  0  0  1  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  1  1  0  0  0  0  0  0  0  1  0  …  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  0  0  0  0  0  0  0  0  1  0     0  0  0  1  0  0  0  0  0  0  0  0  0
 ⋮                 ⋮                 ⋮     ⋱           ⋮                 ⋮
 1  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  1  0  0  0  0  0  0  0  0  0  …  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  2  0  0  0  0  1  1  0  0  0     0  0  0  0  0  0  0  0  0  0  0  1  1
 0  0  0  0  0  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  1  0  0  0  0  0  0  0  1  0     0  0  0  0  0  0  0  0  0  0  0  0  0
 0  0  0  0  2  0  0  0  0  0  0  0  0  0     0  0  0  0  0  0  0  0  1  0  0  0  0
```

```
hash_dtm(crps)
```

TF (Term Frequency)

```
14640×12261 SparseArrays.SparseMatrixCSC{Float64,Int64} with 101505 stored entries:
```

```
  [217  ,     1]  =  0.166667
  [1064 ,     1]  =  0.0909091
  [1369 ,     1]  =  0.0769231
  [1750 ,     1]  =  0.125
  [2890 ,     1]  =  0.0909091
  [3055 ,     1]  =  0.0714286
  ⋮
  [11554, 12257]  =  0.333333
  [11596, 12257]  =  0.25
  [11721, 12258]  =  0.2
  [11703, 12259]  =  0.25
  [11707, 12259]  =  0.25
  [5395 , 12260]  =  0.1
  [6999 , 12261]  =  0.333333
```

```
tf(DocumentTermMatrix(crps))
```

TF-IDF (Term Frequency - Inverse Document Frequency)

```
14640×12261 SparseArrays.SparseMatrixCSC{Float64,Int64} with 101505 stored entries:
  [217  ,     1]  =  1.13649
  [1064 ,     1]  =  0.619902
  [1369 ,     1]  =  0.524533
  [1750 ,     1]  =  0.852366
  [2890 ,     1]  =  0.619902
  [3055 ,     1]  =  0.487066
  ⋮
  [11554, 12257]  =  2.73507
  [11596, 12257]  =  2.0513
  [11721, 12258]  =  1.9183
  [11703, 12259]  =  2.22459
  [11707, 12259]  =  2.22459
  [5395 , 12260]  =  0.959151
  [6999 , 12261]  =  3.19717
```

```
tf_idf(DocumentTermMatrix(crps))
```

Using Navie Bayes Classifier

To learn about the classifier, I have copied the following sample code from **this TextAnalysis.jl doc string**.

```
using TextAnalysis: NaiveBayesClassifier, fit!, predict
```

```
(:spam ⇒ 0.5988304093567252, :non_spam ⇒ 0.4011695906432749)
```

```
let m = NaiveBayesClassifier([:spam, :non_spam])
    fit!(m, "this is spam", :spam)
    fit!(m, "this is not spam", :non_spam)
    predict(m, "is this a spam")
end
```

```
(:spam ⇒ 0.39506172839506176, :non_spam ⇒ 0.6049382716049383)
```

```
let m = NaiveBayesClassifier([:spam, :non_spam])
    fit!(m, "this is spam", :spam)
    fit!(m, "this is not spam", :non_spam)
    predict(m, "I'm not spam")
end
```

Let's build our own classifier.

In our data frame, we already have a column `x_string_doc` with `StringDocuments` values. So we can just fit them to the classifier.

```
model = let
    classes = unique(df2.airline_sentiment)
    nbc = NaiveBayesClassifier(classes)
    for (sd, class) in zip(df2.x_string_doc , df2.airline_sentiment)
        fit!(nbc, sd, class)
    end
    nbc
end;
```

Let's rock and roll!

```
function test_model(predictor)
    df = DataFrame(text = [
        "whatever airline sucks!",
        "i love @virginamerica service :-)",
        "just ok",
        "hello world"])

    df.analysis = predictor.(df.text)

    df.positive = getindex.(df.analysis, "positive")
    df.negative = getindex.(df.analysis, "negative")
    df.neutral = getindex.(df.analysis, "neutral")

    select!(df, Not(:analysis))

    return df
end;
```

4 rows × 4 columns

	text String	positive Float64	negative Float64	neutral Float64
1	whatever airline sucks!	0.086403	0.876043	0.0375539
2	i love @virginamerica service :-)	0.902868	0.0257353	0.0713966
3	just ok	0.448319	0.216309	0.335373
4	hello world	0.178111	0.121133	0.700756

```
test_model(x -> predict(model, tweet_string_doc(x)))
```

Using features (words)

Since we have already generated 1-gram, I wonder if it could make the training faster.

The `NaiveBayesClassifier` comes with another constructor that takes a vector of words to initize the underlying array. This can be found easily from the lexicon of the corpus.

```
12261
```

```
lexicon(crps) |> keys |> length
```

I realized that the TextAnalysis.jl package currently does not provide a `fit!` function for `NGramDocument`. Let's define a patch here just for fun!

```
function TextAnalysis.fit!(c::NaiveBayesClassifier, ngd::NGramDocument, class)
    fs = ngrams(ngd)
    for k in keys(fs)
        k in c.dict || extend!(c, k)
    end
    fit!(c, TextAnalysis.features(fs, c.dict), class)
end
```

Creating an ngram model is fairly straightforward.

```
ngrams_model = let
    words = collect(keys(lexicon(crps)))
    classes = unique(df2.airline_sentiment)
    nbc = NaiveBayesClassifier(words, classes)
    for (ngd, class) in zip(df2.x_ngrams , df2.airline_sentiment)
        fit!(nbc, ngd, class)
    end
    nbc
end;
```

For testing, define a predictor function that takes any string (`x`) and it would determine its features and call the `predict` function with the model.)

```
function ngram_predictor(model)
    features(x) = TextAnalysis.features(ngrams(tweet_string_doc(x)),
ngrams_model.dict)
    return x -> predict(model, features(x))
end;
```

Let's rock and roll!

4 rows × 4 columns

	text String	positive Float64	negative Float64	neutral Float64
1	whatever airline sucks!	0.086403	0.876043	0.0375539
2	i love @virginamerica service :-)	0.902868	0.0257353	0.0713966
3	just ok	0.448319	0.216309	0.335373

	text String	positive Float64	negative Float64	neutral Float64
4	hello world	0.178111	0.121133	0.700756

```
test_model(ngram_predictor(ngrams_model))
```

Let's compare with the result from using `StringDocument`. As expected, they are the same.

4 rows × 4 columns

	text String	positive Float64	negative Float64	neutral Float64
1	whatever airline sucks!	0.086403	0.876043	0.0375539
2	i love @virginamerica service :-)	0.902868	0.0257353	0.0713966
3	just ok	0.448319	0.216309	0.335373
4	hello world	0.178111	0.121133	0.700756

```
test_model(x -> predict(model, tweet_string_doc(x)))
```

Determining accuracy

What well does the Naive Bayes Classifier work?

As the `predict` function returns a `Dict` object with the probabilities assigned to each class, we need to have choose the best option. Let's define a function for that.

```
function predict_and_choose(c::NaiveBayesClassifier, sd::StringDocument)
    val = predict(c, sd)
    return argmax(val)
end;
```

Now, make prediction over all 14K tweets.

```
df2.yhat = predict_and_choose.(Ref(model), df2.x_string_doc);
```

```
hits = 12331
```

```
hits = count(df2.airline_sentiment .== df2.yhat)
```

```
misses = 2309
```

```
misses = nrow(df2) - hits
```

```
wayoff = 700
```

```
wayoff = count(
            (df2.airline_sentiment .!== df2.yhat) .&
```

```
            (df2.airline_sentiment .!== "neutral") .&
            (df2.yhat .!== "neutral"))
```

```
accuracy_percentage = 84.22814207650273
```

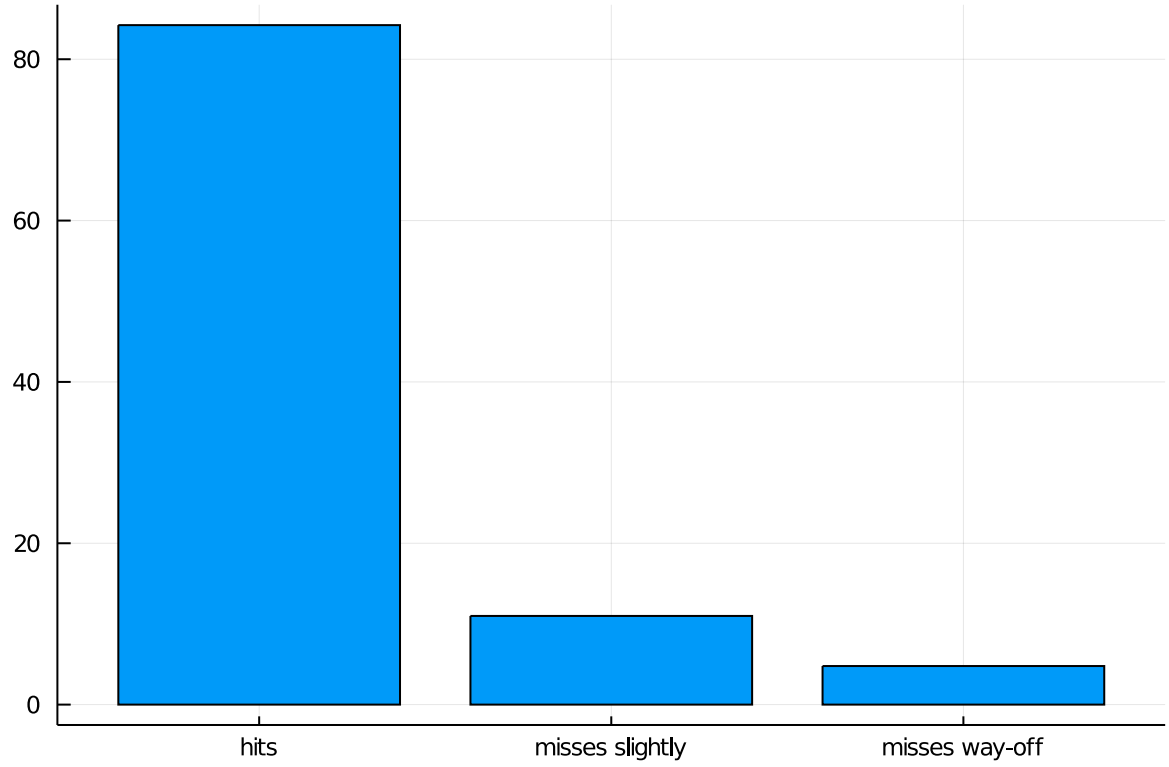
```
accuracy_percentage = hits / (hits + misses) * 100
```

```
slightly_off_percentage = 10.990437158469945
```

```
slightly_off_percentage = (misses - wayoff) / (hits + misses) * 100
```

```
way_off_percentage = 4.781420765027322
```

```
way_off_percentage = wayoff / (hits + misses) * 100
```




```
bar(["hits", "misses slightly", "misses way-off"],
    [accuracy_percentage, slightly_off_percentage , way_off_percentage];
    legend = :none)
```

To-do's

Things that I'd like to do but don't have time at the moment.

1. Try 2-grams and see if it predicts better. However, this will explode the number of features a lot. Just doing 1-gram already gives 12K features.
2. Stemming will probably reduce the size. Not sure how it affects predictive power.
3. Splitting into train/test sets and do cross validations.