1. **Classification of political speeches.** You are given the text of 3119 speeches made by presidential candidates between 1996 and 2016. (The collection of speeches is called the corpus.) Each speech was either made by a Democrat (D) or a Republican (R). You will learn a classifier that can identify a candidate’s political affiliation based on their speech.

The file `speeches.json` contains $U$, a list of all the speeches; $V$, the list of the corresponding party affiliations; $\text{words}$, a list of every word used at least once in one of the speeches; and $\text{freqs}$, a list of how many times each word was used in the speeches.

The $i$th speech in the corpus is $u_i$, and the corresponding affiliation is $v_i \in \{D, R\}$. You will need to utilize the `LinearAlgebra`, `Statistics`, `DataStructures`, and `Flux` packages.

We will be utilizing a particular feature transformation, the term-frequency inverse document frequency (TFIDF) embedding, which is a metric of how important a word is in a corpus. (We will be using this feature transformation along with other standard feature transformations, i.e., appending a constant term and standardization.) We define the term-frequency embedding as

\[
T^{TF}(u)_j = \frac{\text{number of occurrences of word } j \text{ in } u}{\text{the number of words in } u}
\]

This maps a speech $u$ to a vector in $\mathbb{R}^{n_{\text{words}}}$, where $n_{\text{words}}$ is the number of words in the corpus, which we take as the top 2000 most frequent words (so $n_{\text{words}} = 2000$).

From this, we construct the TFIDF embedding. The document frequency of word $j$ is

\[
f_{\text{doc}}(j) = \frac{\text{the number of documents in which the word occurs}}{n}
\]

and the TFIDF embedding is

\[
T^{\text{TFIDF}}(u)_j = T^{TF}(u)_j \log(1 + 1/f_{\text{doc}}(j))
\]

The file `utils.jl` contains utility functions you may find useful. In particular, the functions $\text{TFIDF}(U, \text{words}, \text{freqs})$ and $\text{standardizeplusone}(U, \text{means}, \text{stds})$ will help with the input embeddings. $\text{multi-logistic}(X, Y, \text{reps}; \lambda)$ will solve the multi-class classification problem; the inputs are training input and output data $X$ and $Y$, a list of the representations of the embedded outputs $\text{reps}$, and an optional regularization hyper-parameter term $\lambda$, which defaults to 1. The function outputs the parameterized predictor $\text{predicty}$ for the multi-class classifier, and the model parameters $\theta$. (Since $\text{predicty}$ is a predictor, its argument is simply $x$; to get an (embedded) prediction $\hat{y}$ from a data input $x$, use $\hat{y} = \text{predicty}(x)$.)
Randomly partition 80% of the data into a training set, and 20% into a test set. Using the embedding

\[ x = \phi(u) = (1, \text{standardize}(T_{\text{TFIDF}}(u))) \]

i.e., by standardizing the TFIDF-embedded vector and appending a constant feature to it (so \( d = n_{\text{words}} + 1 = 2001 \)).

The embedded output \( y = \psi(v) \) embeds the party affiliations; we will use \( \psi : \{D, R\} \to \mathbb{R} \) such that \( \psi(D) = 0 \) and \( \psi(R) = 1 \). Embed the raw training and test outputs in this way.

(a) Inspect the `multi_logistic` function in `utils.jl`. What is the loss function used? What is the regularization function used (if any)? A simple response, using no math, is sufficient (e.g., “SVM loss with \( \ell_1 \)-regularization”).

(b) For the regularization hyper-parameter \( \lambda \in \{10^{-2}, 10^{-1}, 10^0, 10^1, 10^2\} \), train a classifier on the training set, using the `multi_logistic` function in `utils.jl`. Report the test accuracy, and highlight the value of \( \lambda \) that gave you the best test accuracy. In addition, report the \( 2 \times 2 \) confusion matrix on the test data.

(c) Using the model that gave you the best test accuracy, analyze the weights \( \theta_{2,k} \) and find the five most useful words for classification. Please leave out the first weight \( \theta_1 \) as it corresponds to the offset.

_Hint._ For part (c), use the absolute value of the weights to determine the importance of a word to the classifier.

_Julia hints._

- For accuracy, you may use the following function in Julia:

  \[
  \text{accuracy}(a, b) = \text{Statistics.mean}(a .== b),
  \]

  where \( a \) and \( b \) are arrays.

- The function `TFIDF(U, words, freqs)` has two outputs: the TFIDF-transformed feature vector, and a dictionary corresponding to the words used in the corpus. The second output will be useful for part (c).

- For part (b), training may take several minutes.

- In part (c), we recommend using the `sortperm` function, which returns the indices of a vector sorted from smallest to largest. If we call `sortperm([3, 1, 2])` we get back \([2, 3, 1]\). If we call `sortperm([3, 1, 2], rev=true)` we get back \([1, 3, 2]\). Call `sortperm` with `rev=true` to find the indices of the largest absolute values of theta. Then you can index into the dictionary returned by `TFIDF` to retrieve the most important words.

_Solution._

(a) The loss function used is logistic loss and the regularization is sum of squares.
(b) We find that $\lambda = 1$ performs the best, yielding an accuracy of approximately 89% on the test set.

(c) We found that the 5 most important words were

\{\text{Republicans, Speaker, Carolina, Madam, Democrat}\}.
In [1]:

```plaintext
using Random
using DataStructures
using LinearAlgebra
using Statistics
using Flux

include("utils.jl")
include("readclassjson.jl")
# include("writeclassjson.jl")
```

Out[1]: readclassjson (generic function with 1 method)

In [2]:

```plaintext
function getstatistics(U)
    means = [Statistics.mean(x) for x in eachcol(U)]
    stds = [Statistics.std(x) for x in eachcol(U)]
    return means, stds
end
```

Out[2]: getstatistics (generic function with 1 method)

In [ ]:

```plaintext
# import data
data = readclassjson("speeches.json")
U, v, words, freqs = data["U"], data["V"], data["words"], data["freqs"]
```

In [ ]:

```plaintext
# convenient function that takes a trained model => accuracy
accuracy(y, y_hat) = mean( y .== y_hat )

X, top_words = TFIDF(U, words, freqs)

# convert D -> -1, R -> 1
embed_output(s::String) = (s == "D") ? [0] : [1]
embed_output(v::Array{String, 1}) = embed_output.(v)
y = hcat(embed_output(v)...)
```

In [52]:

```plaintext
train_idx = 2500

X_train = X[1:train_idx, :]
y_train = y[1:train_idx, :]

X_test = X[train_idx+1:end, :]
y_test = y[train_idx+1:end, :]

means, stds = getstatistics(X_train)
X_train = standardize_plus_one(X_train,means,stds)
X_test = standardize_plus_one(X_test,means,stds)

# reps = [[1,0], [0,1]]
reps = [[0.0], [1.0]]
```

Out[52]:

```
2-element Vector{Vector{Float64}}:
  [0.0]
  [1.0]
```
In [53]:
# lambdas = 10 .^ range(-2, stop=2, length=5)
# acc_train = []
# acc_test = []
# models = []
# for lam in lambdas
#     println("LAM=",lam)
#     predicty, theta = multi_logistic(X_train, y_train, reps, lambda=lambda)
#     y_pred_train = [predicty(X_train[i,:])[2] >= 0 for i = 1:size(X_train,1)]
#     push!(acc_train, accuracy(2y_train[:,2].-1, sign.(y_pred_train)))
#     y_pred_test = [predicty(X_test[i,:])[2] >= 0 for i = 1:size(X_test,1)]
#     push!(acc_test, accuracy(2y_test[:,2].-1, sign.(y_pred_test)))
#     push!(models, [predicty, theta])
#     println(acc_train, acc_test)
# end

In [54]:
# best_predicty, best_theta = models[argmax(acc_test)]
best_predicty, best_theta = multi_logistic(X_train, y_train, reps, lambda=1)
y_pred_test = [best_predicty(X_test[i,:])[1] > 0 for i = 1:size(X_test,1)]
accuracy(y_test, sign.(y_pred_test))

In [55]:
function confusion_matrix(model, U, v, classes)
    K = length(classes)  # Number of classes
    C = zeros(Int, K, K)  # we'll fill this in
    for i=1:K  # Loop over ROWS
        for j=1:K  # Loop over COLUMNS
            C[i,j] = sum([model(U[k,:]) == classes[i] && v[k] == classes[j] for k=1:size(v)[1]])
        end
    end
    return C
end

function model(x)
    return best_predicty(x)[1] > 0
end
# Hint: Be careful of the difference between [0.0] and 0.0
confusion_matrix(model, X_test, y_test, [0, 1])

Out[55]:
2×2 Matrix{Int64}:
  46  56
  7 510
#### Part c

```python
in [56]: top_words[sortperm(abs.(best_theta[2:end]), rev=true)[1:6]]
```

```
Out[56]: 6-element Vector{String}:
    "Republicans"
    "Speaker,"
    "And"
    "Madam"
    "Democrat"
    "Carolina"
```

```python
In [57]: best_theta
```

```
Out[57]: 2001×1 Matrix{Float64}:
    -0.016838820069920178
    0.01532621117993073
    -0.006703047089183853
    -0.006823483559282118
    -0.006834468338578208
    0.03475682932712811
    0.011837034578991264
    0.003814928407227209
    0.006804370040480136
    0.008044146615251348
    -0.008913700592955237
    -0.010696915445250326
    ⋮
    -0.008148581751386883
    -0.015020236306508774
    -0.010774910396823427
    -0.013399482924468294
    0.0024661971356174965
    0.014633224160524393
    -0.014584242063655283
    -0.0031436991611190615
    -0.12924557094451155
    -0.00728145826435716
    -0.002887658579781412
    -0.00798259254600218
```
2. Probabilistic classification on the iris dataset.

Commencement is around the corner, and with it, celebratory flowers! However, some of the flowers are difficult to tell apart. What can we do? In this problem you will build a probabilistic classifier for the well-known Iris Dataset, which contains measurements of three similar-looking irises\(^1\).

You have been provided a function in `probabilistic_classification.jl`:

```
probabilistic_classification(X, y, numiters=500)
```

where \(X\) is a matrix of features, \(y\) is a vector of labels, and \(\text{numiters}\) controls how many epochs Flux runs (you can decrease it if your code is taking too long to train). The return value is a function \text{model} defined so \text{model}(x) \text{ returns } y_{\text{hat}}.

(a) Inspect the `probabilistic_classification` function. How many parameters are being trained? What loss function is being used? (You can just provide the name of the loss function as opposed to an equation for it.)

(b) Now we will import the data. Instead of using a JSON file, we will import the Iris dataset from a repository of common datasets: `MLDatasets.jl` To access the data, write `using MLDatasets` at the top of your code. Then you can access `Iris.features()` and `Iris.labels()`. Convert the labels, which are name strings, into a one-hot encoding.

Partition your data with an 80/20 train/test split and use `probabilistic_classification` to train your model.

First, we will use the probabilistic classifier as a predictor. Define a new function \text{predictor} that calls your model and converts the probabilistic result to a label. Report the accuracy of your classifier on the train and test sets.

(c) Define a function to compute the average negative log likelihood:

\[
\mathcal{L} = -\frac{1}{n} \sum_{i=1}^{n} \log (\hat{p}^{(i)}(v^{(i)}))
\]

where \(\hat{p}^{(i)}(v^{(i)})\) is the predicted probability of the correct label \(v^{(i)}\).

Do we want \(\mathcal{L}\) to be large or small?

Compute \(\mathcal{L}\) for your classifier and report its value on the train and test sets.

\textbf{Hint}: You can use a dictionary to map the iris names to numbers. To initialize a dictionary with keys "a":1 and "b":2, use `Dict(a=>1, b=>2)`.

\(^1\)You can see them here [http://www.lac.inpe.br/rafael.santos/Docs/CAP394/WholeStory-Iris.html](http://www.lac.inpe.br/rafael.santos/Docs/CAP394/WholeStory-Iris.html)
Solution.

(a) This classifier uses $Wx + b$. $W$ has size $K \times d$ where $K$ is the number of classes and $d$ is the number of columns in $X$ (number of features). There are 4 features and 3 classes so $W$ has size 12. $b$ has size $K = 3$.

The total number of parameters is 15.

The cross-entropy loss function is used.

(b) Code is below.

(c) The test accuracy we found was 93%. The train accuracy was 95%. It’s possible to see 100% test accuracy due to the random train/test split and small size of the test dataset.

(d) We found the average negative log likelihood on the test set was 0.102 and on the train set was 0.09. The negative log likelihood is a loss function, so we want it to be small. Some students got larger values.
In [1]: #using Pkg
#Pkg.add("MLDatasets")
using Plots, LinearAlgebra, MLDatasets, Flux, Random
include("probabilistic_classification.jl")

In [2]: N = 150
indices = shuffle(1:N)
N_train = (N÷10)*8
N_test = N-N_train
K = 3

X_train = Iris.features()[indices[1:N_train],:]
X_test = Iris.features()[indices[N_train+1:end],:]
y_train = Iris.labels()[indices[1:N_train]]
y_test = Iris.labels()[indices[N_train+1:end]]

labels_to_int = Dict("Iris-setosa" => 1, "Iris-versicolor"=>2, "Iris-virginica"=>3)
y_train_onehot = Flux.onehotbatch([labels_to_int[yi] for yi in y_train], 1:K)
y_test_onehot = Flux.onehotbatch([labels_to_int[yi] for yi in y_test], 1:K)

reps = [float.(Flux.onehot(i, 1:K)) for i=1:K]

Out[2]: 3-element Vector{Vector{Float64}}:
    [1.0, 0.0, 0.0]
    [0.0, 1.0, 0.0]
    [0.0, 0.0, 1.0]

In [3]: model = probabilistic_classification(X_train, y_train_onehot)

In [4]: function predictor(x)
    return argmax(model(x))
end

function accuracy(predictor, X, y)
    preds = [predictor(X[i, :]) for i=1:size(X)[1]]
    return sum(preds .== y)/size(preds)[1]
end

println("Test accuracy: ", accuracy(predictor, X_test, Flux.onecold(y_test_onehot')))
println("Train accuracy: ", accuracy(predictor, X_train, Flux.onecold(y_train_onehot')))

Test accuracy:  1.0
Train accuracy: 0.9666666666666667
In [13]:
    function nl_likelihood(model, X, y)
        N = size(y)[1]
        return -1.0/N * sum( [log.( model(X[i,:])[y[i]] ) for i=1:N] )
    end

    nl_likelihood(model, X_test, Flux.onecold(y_test_onehot'))

Out[13]: 0.046510441546934816

In [14]:
    nl_likelihood(model, X_train, Flux.onecold(y_train_onehot'))

Out[14]: 0.10701865932619203