Homework 2: SVMs, Kernels & Logistic Regression

Due: Tuesday, Oct 15, 2024 at 11:59AM EST

Instructions: Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. It's preferred that you write your answers using software that typesets mathematics (e.g.LaTeX, LyX, or MathJax via iPython), though if you need to you may scan handwritten work. You may find the minted package convenient for including source code in your LaTeX document. If you are using LyX, then the listings package tends to work better.

---

1 Support Vector Machines: SVMs with Pegasos

In this first problem we will use Support Vector Machines to predict whether the sentiment of a movie review was *positive* or *negative*. We will represent each review by a vector $\boldsymbol{x} \in \mathbb{R}^d$ where d is the size of the word dictionary and x_i is equal to the number of occurrence of the i-th word in the review $\boldsymbol{x}$. The corresponding label is either $y = 1$ for a positive review or $y = -1$ for a negative review. In class we have seen how to transform the SVM training objective into a quadratic program using the dual formulation. Here we will use a gradient descent algorithm instead.

Subgradients

Recall that a vector $g \in \mathbb{R}^d$ is a *subgradient* of $f : \mathbb{R}^d \to \mathbb{R}$ at $\boldsymbol{x}$ if for all $\boldsymbol{z}$,

$$f(\boldsymbol{z}) \geq f(\boldsymbol{x}) + g^T(\boldsymbol{z} - \boldsymbol{x}).$$

There may be 0, 1, or infinitely many subgradients at any point. The *subdifferential* of f at a point $\boldsymbol{x}$, denoted $\partial f(\boldsymbol{x})$, is the set of all subgradients of f at $\boldsymbol{x}$. A good reference for subgradients are the course notes on Subgradients by Boyd et al. Below we derive a property that will make our life easier for finding a subgradient of the hinge loss.

1. Suppose $f_1, \ldots, f_m : \mathbb{R}^d \to \mathbb{R}$ are convex functions, and $f(\boldsymbol{x}) = \max_{i=1,\ldots,m} f_i(\boldsymbol{x})$. Let k be any index for which $f_k(\boldsymbol{x}) = f(\boldsymbol{x})$, and choose $g \in \partial f_k((\boldsymbol{x})$ (a convex function on $\mathbb{R}^d$ has a non-empty subdifferential at all points). Show that $g \in \partial f(\boldsymbol{x})$.

2. Give a subgradient of the hinge loss objective $J(\boldsymbol{w}) = \max\left\{0, 1 - y\boldsymbol{w}^T\boldsymbol{x}\right\}$.

SVM with the Pegasos algorithm

You will train a Support Vector Machine using the Pegasos algorithm [1]. Recall the SVM objective using a linear predictor $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$ and the hinge loss:

$$\min_{\boldsymbol{w} \in \mathbb{R}^d} \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \frac{1}{n}\sum_{i=1}^{n} \max\left\{0, 1 - y_i\boldsymbol{w}^T\boldsymbol{x}_i\right\},$$

where n is the number of training examples and d the size of the dictionary. Note that, for simplicity, we are leaving off the bias term b. Note also that we are using ℓ_2 regularization with a parameter λ. Pegasos is stochastic subgradient descent using a step size rule $\eta_t = 1/(\lambda t)$ for iteration number t. The pseudocode is given below:

[1] Shalev-Shwartz et al. Pegasos: Primal Estimated sub-GrAdient SOlver for SVM.

Input: $\lambda > 0$. Choose $w_1 = 0, t = 0$
While termination condition not met
  For $j = 1, \ldots, n$ (assumes data is randomly permuted)
    $t = t + 1$
    $\eta_t = 1/(t\lambda)$;
    If $y_j w_t^T x_j < 1$
      $w_{t+1} = (1 - \eta_t \lambda) w_t + \eta_t y_j x_j$
    Else
      $w_{t+1} = (1 - \eta_t \lambda) w_t$

---

3. Consider the SVM objective function for a single training point[2]: $J_i(\boldsymbol{w}) = \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \max\left\{0, 1 - y_i \boldsymbol{w}^T \boldsymbol{x}_i\right\}$. The function $J_i(\boldsymbol{w})$ is not differentiable everywhere. Specify where the gradient of $J_i(w)$ is not defined. Give an expression for the gradient where it is defined.

4. Show that a subgradient of $J_i(w)$ is given by

$$\boldsymbol{gw} = \begin{cases} \lambda \boldsymbol{w} - y_i \boldsymbol{x}_i & \text{for } y_i \boldsymbol{w}^T \boldsymbol{x}_i < 1 \\ \lambda \boldsymbol{w} & \text{for } y_i \boldsymbol{w}^T \boldsymbol{x}_i \geq 1. \end{cases}$$

   You may use the following facts without proof: 1) If $f_1, \ldots, f_n : \mathbb{R}^d \to \mathbb{R}$ are convex functions and $f = f_1 + \cdots + f_n$, then $\partial f(\boldsymbol{x}) = \partial f_1(\boldsymbol{x}) + \cdots + \partial f_n(\boldsymbol{x})$. 2) For $\alpha \geq 0$, $\partial(\alpha f)(x) = \alpha \partial f(x)$. (Hint: Use the first part of this problem.)

Convince yourself that if your step size rule is $\eta_t = 1/(\lambda t)$, then doing SGD with the subgradient direction from the previous question is the same as given in the pseudocode.

Dataset and sparse representation

We will be using the Polarity Dataset v2.0, constructed by Pang and Lee, provided in the `data_reviews` folder. It has the full text from 2000 movies reviews: 1000 reviews are classified as *positive* and 1000 as *negative*. Our goal is to predict whether a review has positive or negative sentiment from the text of the review. Each review is stored in a separate file: the positive reviews are in a folder called "pos", and the negative reviews are in "neg". We have provided some code in `utils_svm_reviews.py` to assist with reading these files. The code removes some special symbols from the reviews and shuffles the data. Load all the data to have an idea of what it looks like.

A usual method to represent text documents in machine learning is with *bag-of-words*. As hinted above, here every possible word in the dictionary is a feature, and the value of a word feature for a given text is the number of times that word appears in the text. As most words will not appear in any particular document, many of these counts will be zero. Rather than storing many zeros, we use a *sparse representation*, in which only the nonzero counts are tracked. The counts are stored in a key/value data structure, such as a dictionary in Python. For example, "Harry Potter and Harry Potter II" would be represented as the following Python dict: `x={'Harry':2, 'Potter':2, 'and':1, 'II':1}`.

5. Write a function that converts an example (a list of words) into a sparse bag-of-words

[2]Recall that if i is selected uniformly from the set $\{1, \ldots, n\}$, then this objective function has the same expected value as the full SVM objective function.

representation. You may find Python's Counter[3] class to be useful here. Note that a Counter is itself a dictionary.

6. Load all the data and split it into 1500 training examples and 500 validation examples. Format the training data as a list `X_train` of dictionaries and `y_train` as the list of corresponding 1 or -1 labels. Format the test set similarly.

We will be using linear classifiers of the form $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$, and we can store the $\boldsymbol{w}$ vector in a sparse format as well, such as `w={'minimal':1.3, 'Harry':-1.1, 'viable':-4.2, 'and':2.2, 'product':9.1}`. The inner product between $\boldsymbol{w}$ and $\boldsymbol{x}$ would only involve the features that appear in both `x` and `w`, since whatever doesn't appear is assumed to be zero. For this example, the inner product would be `x(Harry) * w(Harry) + x(and) * w(and) = 2*(-1.1) + 1*(2.2)`. To help you along, `utils_svm_reviews.py` includes two functions for working with sparse vectors: 1) a dot product between two vectors represented as dictionaries and 2) a function that increments one sparse vector by a scaled multiple of another vector, which is a very common operation. It is worth reading the code, even if you intend to implement it yourself. You may get some ideas on how to make things faster.

7. Implement the Pegasos algorithm to run on a sparse data representation. The output should be a sparse weight vector $\boldsymbol{w}$ represented as a dictionary. Note that our Pegasos algorithm starts at $w = 0$, which corresponds to an empty dictionary. Terminate the algorithm when the classification error is within a tolerance of 0.001. **Note:** With this problem, you will need to take some care to code things efficiently. In particular, be aware that making copies of the weight dictionary can slow down your code significantly. If you want to make a copy of your weights (e.g. for checking for convergence), make sure you don't do this more than once per epoch. **Also**: If you normalize your data in some way, be sure not to destroy the sparsity of your data. Anything that starts as 0 should stay at 0.

Note that in every step of the Pegasos algorithm, we rescale every entry of w_t by the factor $(1 - \eta_t\lambda)$. Implementing this directly with dictionaries is very slow. We can make things significantly faster by representing w as $w = sW$, where $s \in \mathbb{R}$ and $W \in \mathbb{R}^d$. You can start with $s = 1$ and W all zeros (i.e. an empty dictionary). Note that both updates (i.e. whether or not we have a margin error) start with rescaling w_t, which we can do simply by setting $s_{t+1} = (1 - \eta_t\lambda)\, s_t$.

8. If the update is $w_{t+1} = (1 - \eta_t\lambda)w_t + \eta_t y_j x_j$, then verify that the Pegasos update step is equivalent to:

$$\begin{aligned} s_{t+1} &= (1 - \eta_t\lambda)\, s_t \\ W_{t+1} &= W_t + \frac{1}{s_{t+1}}\eta_t y_j x_j. \end{aligned}$$

Implement the Pegasos algorithm with the (s, W) representation described above. [4]

[3] `https://docs.python.org/2/library/collections.html`

[4] There is one subtle issue with the approach described above: if we ever have $1 - \eta_t\lambda = 0$, then $s_{t+1} = 0$, and we'll have a divide by 0 in the calculation for W_{t+1}. This only happens when $\eta_t = 1/\lambda$. With our step-size rule of $\eta_t = 1/(\lambda t)$, it happens exactly when $t = 1$. So one approach is to just start at $t = 2$. More generically, note that if $s_{t+1} = 0$, then $w_{t+1} = 0$. Thus an equivalent representation is $s_{t+1} = 1$ and $W = 0$. Thus if we ever get $s_{t+1} = 0$, simply set it back to 1 and reset W_{t+1} to zero, which is an empty dictionary in a sparse representation.

9. Run both implementations of Pegasos on the training data for a couple epochs. Make sure your implementations are correct by verifying that the two approaches give essentially the same result. Report on the time taken to run each approach.

10. Write a function `classification_error` that takes a sparse weight vector `w`, a list of sparse vectors `X` and the corresponding list of labels `y`, and returns the fraction of errors when predicting y_i using $\text{sign}(\boldsymbol{w}^T \boldsymbol{x}_i)$. In other words, the function reports the 0-1 loss of the linear predictor $f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x}$.

11. Search for the regularization parameter that gives the minimal percent error on your test set. You should now use your faster Pegasos implementation, and run it to convergence. A good search strategy is to start with a set of regularization parameters spanning a broad range of orders of magnitude. Then, continue to zoom in until you're convinced that additional search will not significantly improve your test performance. Plot the test errors you obtained as a function of the parameters λ you tested. (Hint: the error you get with the best regularization should be closer to 15% than 20%. If not, maybe you did not train to convergence.)

Error Analysis

Recall that the *score* is the value of the prediction $f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x}$. We like to think that the magnitude of the score represents the confidence of the prediction. This is something we can directly verify or refute.

12. Break the predictions on the test set into groups based on the score (you can play with the size of the groups to get a result you think is informative). For each group, examine the percentage error. You can make a table or graph. Summarize the results. Is there a correlation between higher magnitude scores and accuracy?

2 Kernel Methods

2.1 Kernelization review

Consider the following optimization problem on a data set $(\boldsymbol{x}_1, y_1), \ldots (\boldsymbol{x}_n, y_n) \in \mathbb{R}^d \times \mathcal{Y}$:

$$\min_{w \in \mathbb{R}^d} R\left(\sqrt{\langle \boldsymbol{w}, \boldsymbol{w} \rangle}\right) + L\left(\langle \boldsymbol{w}, \boldsymbol{x}_1 \rangle, \ldots, \langle \boldsymbol{w}, \boldsymbol{x}_n \rangle\right),$$

where $\boldsymbol{w}, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathbb{R}^d$, and $\langle \cdot, \cdot \rangle$ is the standard inner product on $\mathbb{R}^d$. The function $R : [0, \infty) \to \mathbb{R}$ is nondecreasing and gives us our regularization term, while $L : \mathbb{R}^n \to \mathbb{R}$ is arbitrary[5] and gives us our loss term. We noted in lecture that this general form includes soft-margin SVM and ridge regression, though not lasso regression. Using the representer theorem, we showed if the optimization problem has a solution, there is always a solution of the form $\boldsymbol{w} = \sum_{i=1}^n \boldsymbol{\alpha}_i \boldsymbol{x}_i$,

[5] You may be wondering "Where are the y_i's?". They're built into the function L. For example, a square loss on a training set of size 3 could be represented as $L(s_1, s_2, s_3) = \frac{1}{3}\left[(s_1 - y_1)^2 + (s_2 - y_2)^2 + (s_3 - y_3)^3\right]$, where each s_i stands for the ith prediction $\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle$.

for some $\alpha \in \mathbb{R}^n$. Plugging this into the our original problem, we get the following "kernelized" optimization problem:

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^n} R\left(\sqrt{\boldsymbol{\alpha}^T K \boldsymbol{\alpha}}\right) + L\left(K\boldsymbol{\alpha}\right),$$

where $K \in \mathbb{R}^{n \times n}$ is the Gram matrix (or "kernel matrix") defined by $K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle$. Predictions are given by

$$f(x) = \sum_{i=1}^{n} \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x}),$$

and we can recover the original $\boldsymbol{w} \in \mathbb{R}^d$ by $\boldsymbol{w} = \sum_{i=1}^n \alpha_i \boldsymbol{x}_i$.

The *kernel trick* is to swap out occurrences of the kernel k (and the corresponding Gram matrix K) with another kernel. For example, we could replace $k(x_i, x_j) = \langle x_i, x_j \rangle$ by $k'(x_i, x_j) = \langle \psi(x_i), \psi(x_j) \rangle$ for an arbitrary feature mapping $\boldsymbol{\psi} : \mathbb{R}^d \to \mathbb{R}^D$. In this case, the recovered $\boldsymbol{w} \in \mathbb{R}^D$ would be $\boldsymbol{w} = \sum_{i=1}^n \alpha_i \boldsymbol{\psi}(\boldsymbol{x}_i)$ and predictions would be $\langle \boldsymbol{w}, \boldsymbol{\psi}(\boldsymbol{x}_i) \rangle$.

More interestingly, we can replace k by another kernel $k''(\boldsymbol{x}_i, \boldsymbol{x}_j)$ for which we do not even know or cannot explicitly write down a corresponding feature map $\boldsymbol{\psi}$. Our main example of this is the RBF kernel

$$k(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right),$$

for which the corresponding feature map ψ is infinite dimensional. In this case, we cannot recover w since it would be infinite dimensional. Predictions must be done using $\alpha \in \mathbb{R}^n$, with $f(x) = \sum_{i=1}^n \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x})$.

Your implementation of kernelized methods below should not make any reference to $\boldsymbol{w}$ or to a feature map $\boldsymbol{\psi}$. Your learning routine should return $\boldsymbol{\alpha}$, rather than $\boldsymbol{w}$, and your prediction function should also use $\boldsymbol{\alpha}$ rather than $\boldsymbol{w}$. This will allow us to work with kernels that correspond to infinite-dimensional feature vectors.

Kernels and Kernel Machines

There are many different families of kernels. So far we spoken about linear kernels, RBF/Gaussian kernels, and polynomial kernels. The last two kernel types have parameters. In this section, we'll implement these kernels in a way that will be convenient for implementing our kernelized ridge regression later on. For simplicity, we will assume that our input space is $\mathcal{X} = \mathbb{R}$. This allows us to represent a collection of n inputs in a matrix $X \in \mathbb{R}^{n \times 1}$. You should now refer to the jupyter notebook `skeleton_code_kernels.ipynb`.

13. Write functions that compute the RBF kernel $k_{\mathrm{RBF}(\sigma)}(x, x') = \exp\left(-\|x - x'\|^2 / \left(2\sigma^2\right)\right)$ and the polynomial kernel $k_{\mathrm{poly}(a,d)}(x, x') = (a + \langle x, x' \rangle)^d$. The linear kernel $k_{\mathrm{linear}}(x, x') = \langle x, x' \rangle$, has been done for you in the support code. Your functions should take as input two matrices $W \in \mathbb{R}^{n_1 \times d}$ and $X \in \mathbb{R}^{n_2 \times d}$ and should return a matrix $M \in \mathbb{R}^{n_1 \times n_2}$ where $M_{ij} = k(W_{i\cdot}, X_{j\cdot})$. In words, the (i, j)'th entry of M should be kernel evaluation between w_i (the ith row of W) and x_j (the jth row of X). For the RBF kernel, you may use the scipy function `cdist(X1,X2,'sqeuclidean')` in the package `scipy.spatial.distance`.

14. Use the linear kernel function defined in the code to compute the kernel matrix on the set of points $x_0 \in \mathcal{D}_X = \{-4, -1, 0, 2\}$. Include both the code and the output.

15. Suppose we have the data set $\mathcal{D}_{X,y} = \{(-4, 2), (-1, 0), (0, 3), (2, 5)\}$ (in each set of parentheses, the first number is the value of x_i and the second number the corresponding value

of the target y_i). Then by the representer theorem, the final prediction function will be in the span of the functions $x \mapsto k(x_0, x)$ for $x_0 \in \mathcal{D}_X = \{-4, -1, 0, 2\}$. This set of functions will look quite different depending on the kernel function we use. The set of functions $x \mapsto k_{\text{linear}}(x_0, x)$ for $x_0 \in \mathcal{D}_X$ and for $x \in [-6, 6]$ has been provided for the linear kernel.

(a) Plot the set of functions $x \mapsto k_{\text{poly}(1,3)}(x_0, x)$ for $x_0 \in \mathcal{D}_X$ and for $x \in [-6, 6]$.

(b) Plot the set of functions $x \mapsto k_{\text{RBF}(1)}(x_0, x)$ for $x_0 \in \mathcal{D}_X$ and for $x \in [-6, 6]$.

Note that the values of the parameters of the kernels you should use are given in their definitions in (a) and (b).

16. By the representer theorem, the final prediction function will be of the form $f(x) = \sum_{i=1}^{n} \alpha_i k(x_i, x)$, where $x_1, \ldots, x_n \in \mathcal{X}$ are the inputs in the training set. We will use the class `Kernel_Machine` in the skeleton code to make prediction with different kernels. Complete the `predict` function of the class `Kernel_Machine`. Construct a `Kernel_Machine` object with the RBF kernel (sigma=1), with prototype points at $-1, 0, 1$ and corresponding weights α_i $1, -1, 1$. Plot the resulting function.

Note: It may be helpful to use partial application on your kernel functions. For example, if your polynomial kernel function has signature `polynomial_kernel(W, X, offset, degree)`, you can write `k = functools. partial(polynomial_kernel, offset=2, degree=2)`, and then a call to `k(W,X)` is equivalent to `polynomial_kernel(W, X, offset=2, degree=2)`, the advantage being that the extra parameter settings are built into `k(W,X)`. This can be convenient so that you can have a function that just takes a kernel function `k(W,X)` and doesn't have to worry about the parameter settings for the kernel.

3 Logistic Regression

Consider a binary classification setting with input space $\mathcal{X} = \mathbb{R}^d$, outcome space $\mathcal{Y}_{\pm} = \{-1, 1\}$, and a dataset $\mathcal{D} = \big((x^{(1)}, y^{(1)}), \cdots, (x^{(n)}, y^{(n)})\big)$.

Equivalence of ERM and probabilistic approaches

ERM with logistic loss.
Consider a linear scoring function in the space $\mathcal{F}_{\text{score}} = \big\{x \mapsto x^T w \mid w \in \mathbb{R}^d\big\}$. A simple way to make predictions (similar to what we've seen with the perceptron algorithm) is to predict $\hat{y} = 1$ if $x^T w > 0$, or $\hat{y} = \text{sign}(x^T w)$. Accordingly, we consider margin-based loss functions that relate the loss with the margin, $yx^T w$. A positive margin means that $x^T w$ has the same sign as y, i.e. a correct prediction. Specifically, let's consider the **logistic loss** function $\ell_{\text{logistic}}(y, w) = \log\big(1 + \exp(-yw^T x)\big)$. This is a margin-based loss function that you have now encountered several times. Given the logistic loss, we can now minimize the empirical risk on our dataset $\mathcal{D}$ to obtain an estimate of the parameters, $\hat{w}$.

MLE with a Bernoulli response distribution and the logistic link function.
As discussed in the lecture, given that $p(y = 1 \mid x; w) = 1/(1+\exp(-x^T w))$, we can estimate w by maximizing the likelihood, or equivalently, minimizing the negative log-likelihood ($\text{NLL}_{\mathcal{D}}(\text{w})$ in short) of the data.

17. Show that the two approaches are equivalent, i.e. they will produce the same solution for w.

Linearly Separable Data

In this problem, we will investigate the behavior of MLE for logistic regression when the data is linearly separable.

18. Show that the decision boundary of logistic regression is given by $\{x\colon x^T w = 0\}$. Note that the set will not change if we multiply the weights by some constant c.

19. Suppose the data is linearly separable and by gradient descent/ascent we have reached a decision boundary defined by $\hat{w}$ where all examples are classified correctly. Show that we can always increase the likelihood of the data by multiplying a scalar c on $\hat{w}$, which means that MLE is not well-defined in this case. (Hint: You can show this by taking the derivative of $L(c\hat{w})$ with respect to c, where L is the likelihood function.)

Regularized Logistic Regression

As we've shown in above, when the data is linearly separable, MLE for logistic regression may end up with weights with very large magnitudes. Such a function is prone to overfitting. In this part, we will apply regularization to fix the problem.

The ℓ_2 regularized logistic regression objective function can be defined as

$$\begin{aligned} J_{\text{logistic}}(w) &= \hat{R}_n(w) + \lambda\|w\|^2 \\ &= \frac{1}{n}\sum_{i=1}^{n} \log\left(1 + \exp\left(-y^{(i)} w^T x^{(i)}\right)\right) + \lambda\|w\|^2. \end{aligned}$$

20. Prove that the objective function $J_{\text{logistic}}(w)$ is convex. You may use any facts mentioned in the convex optimization notes.

21. Complete the `f_objective` function in the skeleton code, which computes the objective function for $J_{\text{logistic}}(w)$. (Hint: you may get numerical overflow when computing the exponential literally, e.g. try e^{1000} in Numpy. Make sure to read about the log-sum-exp trick and use the numpy function *logaddexp* to get accurate calculations and to prevent overflow.

22. Complete the `fit_logistic_regression_function` in the skeleton code using the `minimize` function from `scipy.optimize`. Use this function to train a model on the provided data. Make sure to take the appropriate preprocessing steps, such as standardizing the data and adding a column for the bias term.

23. Find the ℓ_2 regularization parameter that minimizes the log-likelihood on the validation set. Plot the log-likelihood for different values of the regularization parameter.

24. [Optional] It seems reasonable to interpret the prediction $f(x) = \phi(w^T x) = 1/(1 + e^{-w^T x})$ as the probability that $y = 1$, for a randomly drawn pair (x, y). Since we only have a finite sample (and we are regularizing, which will bias things a bit) there is a question of how well "calibrated" our predicted probabilities are. Roughly speaking, we say $f(x)$ is well calibrated if we look at all examples (x, y) for which $f(x) \approx 0.7$ and we find that close to

70% of those examples have $y = 1$, as predicted... and then we repeat that for all predicted probabilities in $(0, 1)$. To see how well-calibrated our predicted probabilities are, break the predictions on the validation set into groups based on the predicted probability (you can play with the size of the groups to get a result you think is informative). For each group, examine the percentage of positive labels. You can make a table or graph. Summarize the results. You may get some ideas and references from scikit-learn's discussion.