# CPSC 340 Assignment 4 (due March 13 at 11:55pm)

Name(s) and Student ID(s):

## 1 Gaussian RBFs and Regularization

Unfortunately, in practice we often do not know what basis to use. However, if we have enough data then we can make up for this by using a basis that is flexible enough to model any reasonable function. These may perform poorly if we do not have much data, but can perform almost as well as the optimal basis as the size of the dataset grows. In this question you will explore using Gaussian radial basis functions (Gaussian RBFs), which have this property. These RBFs depend on a parameter $\sigma$, which (like $p$ in the polynomial basis) can be chosen using a validation set. In this question, you will also see how cross-validation allows you to tune parameters of the model on a larger dataset than a strict training/validation split would allow.

### 1.1 Regularization

If you run the demo `python main.py 1`, it will load a dataset and randomly split the training examples into a "train" and a "validation" set (it does this randomly since the data is sorted). It will then search for the best value of $\sigma$ for the RBF basis. Once it has the "best" value of $\sigma$, it re-trains on the entire dataset and reports the training error on the full training set as well as the error on the test set.

A strange behaviour appears: if you run the script more than once it might choose different values of $\sigma$. Sometimes it chooses a large value of $\sigma$ (like 32) that follows the general trend but misses the oscillations. Other times it sets $\sigma = 1$ or $\sigma = 2$, which fits the oscillations better but overfits so achieves a similar test error.[1] In the file `linear_models.py`, complete the class `RegularizedRBF`, that fits the model with L2-regularization. Hand in your code, and report the test error you obtain if you train on the full dataset with $\sigma = 1$ and $\lambda = 10^{-12}$ (a very small value).

[1] This behaviour seems to be dependent on your exact setup. Because the $Z^T Z$ matrix with the RBF matrix is really-badly behaved numerically, different floating-point and matrix-operation implementations will handle this in different ways: in some settings it will actually regularize for you!

## 1.2 Cross-Validation

Even with regularization, the randomization of the training/validation sets has an effect on the value of $\sigma$ that we choose (on some runs it still chooses a large $\sigma$ value). This variability would be reduced if we had a larger "train" and "validation" set, and one way to simulate this is with *cross-validation*. Modify the training/validation procedure to use 10-fold cross-validation to select $\sigma$ (with $\lambda$ fixed at $10^{-12}$). Hand in your code and report how this affects the selection of $\sigma$ compared to the original code.

### 1.3 Cost of Non-Parametric Bases

When dealing with larger datasets, an important issue is the dependence of the computational cost on the number of training examples $n$ and the number of features $d$.

Answer the following questions and briefly justify your answers:

1. What is the cost in big-$\mathcal{O}$ notation of training a linear regression model with Gaussian RBFs on $n$ training examples with $d$ features (for fixed $\sigma$ and $\lambda$)?
2. What is the cost in big-$\mathcal{O}$ notation of classifying $t$ new examples with this model?
3. When is it cheaper to train using Gaussian RBFs than using the original linear basis?
4. When is it cheaper to predict using Gaussian RBFs than using the original linear basis?

# 2 Logistic Regression with Sparse Regularization

If you run `python main.py 2`, it will:

1. Load a binary classification dataset containing a training and a validation set.
2. Standardize the columns of `X`, and add a bias variable (in `utils.load_dataset`).
3. Apply the same transformation to `Xvalidate` (in `utils.load_dataset`).
4. Fit a logistic regression model.
5. Report the number of features selected by the model (number of non-zero regression weights).
6. Report the error on the validation set.

Logistic regression does reasonably well on this dataset, but it uses all the features (even though only the prime-numbered features are relevant) and the validation error is above the minimum achievable for this model (which is 1 percent, if you have enough data and know which features are relevant). In this question, you will modify this demo to use different forms of regularization to improve on these aspects.

Note: your results may vary slightly, depending on your software versions, the exact order you do floating-point operations in, and so on.

## 2.1 L2-Regularization

In `linear_models.py`, you will find a class named `LinearClassifier` that defines the fitting and prediction behaviour of a logistic regression classifier. As with ordinary least squares linear regression, the particular choice of a function object (`fun_obj`) and an optimizer (`optimizer`) will determine the properties of your output model. Your task is to implement a logistic regression classifier that uses L2-regularization on its weights. Go to `fun_obj.py` and complete the `LogisticRegressionLossL2` class. This class' constructor takes an input parameter $\lambda$, the L2 regularization weight. Specifically, while `LogisticRegressionLoss` computes

$$f(w) = \sum_{i=1}^{n} \log(1 + \exp(-y_i w^T x_i)),$$

your new class `LogisticRegressionLossL2` should compute

$$f(w) = \sum_{i=1}^{n} \left[\log(1 + \exp(-y_i w^T x_i))\right] + \frac{\lambda}{2}\|w\|^2$$

and its gradient. Submit your function object code. Using this new code with $\lambda = 1$, report how the following quantities change: (1) the training (classification) error, (2) the validation (classification) error, (3) the number of features used, and (4) the number of gradient descent iterations.

Note: as you may have noticed, `lambda` is a special keyword in Python, so we can't use it as a variable name. Some alternative options: `lammy`, `lamda`, `reg_wt`, $\lambda$ if you feel like typing it, the sheep emoji[2], ....

[2]Harder to insert in LaTeX than you'd like; turns out there are some drawbacks to using software written in 1978.

## 2.2 L1-Regularization and Regularization Path

The L1-regularized logistic regression classifier has the following objective function:

$$f(w) = \sum_{i=1}^{n} \left[\log(1 + \exp(-y_i w^T x_i))\right] + \lambda \|w\|_1.$$

Because the L1 norm isn't differentiable when any elements of $w$ are 0 – and that's *exactly what we want to get* – standard gradient descent isn't going to work well on this objective. There is, though, a similar approach called *proximal gradient descent* that does work here.

This is implemented for you in the `GradientDescentLineSearchProxL1` class inside `optimizers.py`. Note that to use it, you *don't include the L1 penalty in your loss function object*; the optimizer handles that itself.

Write and submit code to instantiate `LinearClassifier` with the correct function object and optimizer for L1-regularization. Using this linear model, obtain solutions for L1-regularized logistic regression with $\lambda = 0.01$, $\lambda = 0.1$, $\lambda = 1$, $\lambda = 10$. Report the following quantities per each value of $\lambda$: (1) the training error, (2) the validation error, (3) the number of features used, and (4) the number of gradient descent iterations.

## 2.3 L0 Regularization

The class `LogisticRegressionLossL0` in `fun_obj.py` contains part of the code needed to implement the *forward selection* algorithm, which approximates the solution with L0-regularization,

$$f(w) = \sum_{i=1}^{n} \left[\log(1 + \exp(-y_i w^T x_i))\right] + \lambda \|w\|_0.$$

The class `LinearClassifierForwardSel` in `linear_models.py` will use a loss function object and an optimizer to perform a forward selection to approximate the best feature set. The `for` loop in its `fit()` method is missing the part where we fit the model using the subset `selected_new`, then compute the score and updates the `min_loss` and `best_feature`. Modify the `for` loop in this code so that it fits the model using only the features `selected_new`, computes the score above using these features, and updates the variables `min_loss` and `best_feature`, as well as `self.total_evals`. Hand in your updated code. Using this new code with $\lambda = 1$, report the training error, validation error, number of features selected, and total optimization steps.

Note that the code differs slightly from what we discussed in class, since we're hard-coding that we include the first (bias) variable. Also, note that for this particular case using the L0-norm with $\lambda = 1$ is using the Akaike Information Criterion (AIC) for variable selection.

Also note that, for numerical reasons, your answers may vary depending on exactly what system and package versions you are using. That is fine.

### 2.4 L2- vs. L1- vs. L0-Regularization

For this problem, the relevant features are the bias variable and the featurs with prime numbers. Given this, explain how each of the 3 regularizers (L2-regularization, L1-regularization, and L0-regularization) performed in terms of false positives for feature selection (a false positive would be when a feature is selected but it is not relevant). And then explain how each method did in terms of false negatives.

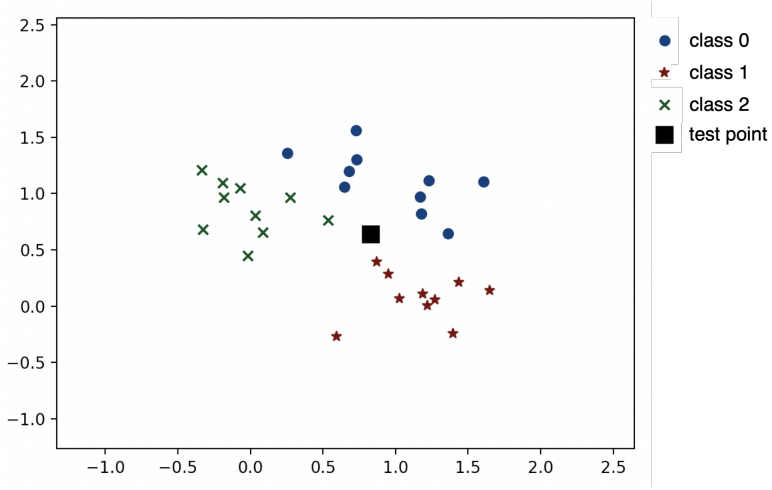
# 3 Multi-Class Logistic Regression

If you run `python main.py 3` the code loads a multi-class classification dataset with $y_i \in \{0, 1, 2, 3, 4\}$ and fits a "one-vs-all" classification model using least squares, then reports the validation error and shows a plot of the data/classifier. The performance on the validation set is ok, but could be much better. For example, this classifier never even predicts that examples will be in classes 0 or 4.

## 3.1 Softmax Classification

Linear classifiers make their decisions by finding the class label $c$ maximizing the quantity $w_c^T x_i$, so we want to train the model to make $w_{y_i}^T x_i$ larger than $w_{c'}^T x_i$ for all the classes $c'$ that are not the true label $y_i$. Here, $c$ is a possible label and $w_{c'}$ is **row** $c'$ of $W$. Similarly, $y_i$ is the training label and $w_{y_i}$ is **row** $y_i$ of $W$. Before we move on to implementing the softmax classifier to fix the issues raised in the introduction, let's do a simple example:

Consider the dataset below, which has 30 training examples, 2 features, and 3 class labels:




Suppose that we want to classify the black square at the location

$$\hat{x} = \begin{bmatrix} +0.84 \\ +0.64 \end{bmatrix}.$$

Suppose that we fit a multi-class linear classifier including bias variable using the softmax loss. We obtain the weight matrix

$$\begin{array}{c} \\ \mathrm{W} = \end{array} \begin{array}{c} \begin{array}{ccc} \text{bias} & x^1 & x^2 \end{array} \\ \begin{bmatrix} +0.00 & +1.62 & +3.47 \\ +4.90 & -3.83 & +0.67 \\ +3.37 & +2.22 & -4.13 \end{bmatrix} \end{array},$$

where the first column corresponds to the bias variable and the last two columns correspond to the two features.

1. What is the meaning of the rows $w_i$ in the matrix $W$?
2. Under this model, what class label would we assign to the test example $\hat{x}$? (show your work)

### 3.2 Softmax Loss

Using a one-vs-all classifier with a least squares objective hurts performance (1) because the classifiers are fit independently (so there is no attempt to calibrate the **rows** of the matrix $W$) and (2) because the squared error loss penalizes the model if it classifies examples "too correctly". An alternative to this model is to use the softmax loss function, which for $n$ training examples is given by

$$f(W) = \sum_{i=1}^{n} \left[ -w_{y_i}^T x_i + \log \left( \sum_{c'=1}^{k} \exp(w_{c'}^T x_i) \right) \right].$$

Derive the partial derivative $\frac{\partial f}{\partial W_{cj}}$ of this loss function with respect to a particular element $W_{cj}$ (the variable in row $c$ and column $j$ of the matrix $W$) – show your work. Try to simplify the derivative as much as possible (but you can express the result in summation notation).

Hint: for the gradient you can use $x_{ij}$ to refer to element $j$ of example $i$. For the first term you will need to separately think about the cases where $c = y_i$ and the cases where $c \neq y_i$. You may find it helpful to use an 'indicator' function, $I(y_i = c)$, which is 1 when $y_i = c$ and is 0 otherwise. Note that you can use the definition of the softmax probability to simplify the second term of the derivative.

### 3.3 Softmax Classifier Implementation

Inside `linear_models.py`, you will find the class `MulticlassLinearClassifier`, which fits $W$ using the softmax loss from the previous section instead of fitting $k$ independent classifiers. As with other linear models, you must implement a function object class in `fun_obj.py`. Find the class named `SoftmaxLoss`. Complete these classes and their methods. Submit your code and report the validation error.

Hint: You may want to use `check_correctness()` to check that your implementation of the gradient is correct.

Hint: With softmax classification, our parameters live in a matrix $W$ instead of a vector $w$. However, most optimization routines (like `scipy.optimize.minimize` or our `optimizers.py`) are set up to optimize with respect to a vector of parameters. The standard approach is to "flatten" the matrix $W$ into a vector (of length $kd$, in this case) before passing it into the optimizer. On the other hand, it's inconvenient to work with the flattened form everywhere in the code; intuitively, we think of it as a matrix $W$ and our code will be more readable if the data structure reflects our thinking. Thus, the approach we recommend is to reshape the parameters back and forth as needed. The skeleton code of `SoftmaxLoss` already has lines reshaping the input vector $w$ into a $k \times d$ matrix using `np.reshape`. You can then compute the gradient using sane, readable code with the $W$ matrix inside `evaluate()`. You'll end up with a gradient that's also a matrix: one partial derivative per element of $W$. Right at the end of `evaluate()`, you can flatten this gradient matrix into a vector using `g.reshape(-1)`. If you do this, the optimizer will be sending in a vector of parameters to `SoftmaxLoss`, and receiving a gradient vector back out, which is the interface it wants – and your `SoftmaxLoss` code will be much more readable, too. You may need to do a bit more reshaping elsewhere, but this is the key piece.

Hint: A naïve implementation of `SoftmaxLoss.evaluate()` might involve many for-loops, which is fine as long as the function and gradient calculations are correct. However, this method might take a very long time! This speed bottleneck is one of Python's shortcomings, which can be addressed by employing pre-computing and lots of vectorized operations. However, it can be difficult to convert your written solutions of $f$ and $g$ into vectorized forms, so you should prioritize getting the implementation to work correctly first. One reasonable path is to first make a correct function and gradient implementation with lots of loops, then (if you want) pulling bits out of the loops into meaningful variables, and then thinking about how you can compute each of the variables in a vectorized way. Our solution code doesn't contain any loops, but the solution code for previous instances of the course actually did; it's totally okay for this course to not be allergic to Python `for` loops.

### 3.4 Cost of Multinomial Logistic Regression

Assuming that we have

- $n$ training examples.
- $d$ features.
- $k$ classes.
- $t$ testing examples.
- $T$ iterations of gradient descent for training.

1. In big-$\mathcal{O}$ notation, what is the cost of training the softmax classifier (briefly justify your answer)?
2. In big-$\mathcal{O}$ notation, what is the cost of classifying the test examples (briefly justify your answer)?

# 4 Very-Short Answer Questions

1. If we fit a linear regression model and then remove all features whose associated weight is small, why is this an ineffective way of performing feature selection?
2. Given 3 features $\{f_1, f_2, f_3\}$, provide an argument that illustrates why the forward selection algorithm is not guaranteed to find an optimal subset of features.
3. What is a setting where you would use the L1-loss, and what is a setting where you would use L1-regularization?
4. Among L0-regularization, L1-regularization, and L2-regularization: which yield convex objectives? Which yield unique solutions? Which yield sparse solutions?
5. What is the effect of $\lambda$ in L1-regularization on the sparsity level of the solution? What is the effect of $\lambda$ on the two parts of the fundamental trade-off?
6. Suppose you have a feature selection method that tends not generate false positives but has many false negatives (it misses relevant variables). Describe an ensemble method for feature selection that could improve the performance of this method.
7. How does the hyper-parameter $\sigma$ affect the shape of the Gaussian RBFs bumps? How does it affect the fundamental tradeoff?
8. What is the main problem with using least squares to fit a linear model for binary classification?
9. Suppose a binary classification dataset has 3 features. If this dataset is "linearly separable", what does this precisely mean in three-dimensional space?
10. Why do we not minimize $\max(0, -y_i w^\top x_i)$ when we fit a binary linear classifier, even though it's a convex approximation to the 0-1 loss?
11. For a linearly-separable binary classification problem, how does an SVM classifier differ from a classifier found using the perceptron algorithm?
12. Which of the following methods produce linear classifiers? (a) binary least squares as in Question 3, (b) the perceptron algorithm, (c) SVMs, (d) logistic regression, and (e) KNN with $k = 1$ and $n = 2$.
13. Why do we use the polynomial kernel to implement the polynomial basis when $d$ and $p$ (degree of polynomial) are large?
14. What is the relationship between the softmax loss and the softmax function?