Problem Set 3: Classification & Convex Optimization

Due: Tuesday, March 24, 2026 at 11:59pm ET

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 or MathJax in 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. For the coding problems, the text **Submission:** indicates all you need to submit in your PDF submission.

Problem 1: Logistic Regression (25 points)

Note: Parts of this problem will be covered in Week 7 (MLE & Conditional Probability Models).

In this problem, we will be exploring *logistic regression*, a predominant method of classification in "classical" machine learning. Throughout, suppose we are in the binary classification setting, with input space $\mathcal{X} = \mathbb{R}^d$ and outcome space $\mathcal{Y} = \{-1, +1\}$. In lecture and lab, we discussed two different ways to end up with logistic regression. The following is review. Throughout this section, the *sigmoid* function will keep popping up:

$$\sigma(z) = \frac{1}{1 + e^{-z}} = (1 + e^{-z})^{-1}.$$

Approach 1: ERM. Suppose we allow ourselves the action space $\mathcal{A} = \mathbb{R}$ and we consider the hypothesis class of linear functions $\mathcal{H} = \{h(x) = w^\top x : w \in \mathbb{R}^d\}$. With this hypothesis class, recall that we defined the *margin* on an example (x, y) as the product of our predicted action and the true value, y. That is, the margin on (x, y) is $m = h(x)y = (w^\top x)y$. We consider the margin-based *logistic loss function*:

$$\ell_{\log}(m) = \log(1 + e^{-m}).$$

If we perform ERM over a training dataset $D = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$ with respect to this loss function for the class of linear functions, we obtain the objective:

$$\hat{R}_n(w) = \frac{1}{n}\sum_{i=1}^{n} \ell_{\log}(y^{(i)} w^\top x^{(i)}) = \frac{1}{n}\sum_{i=1}^{n} \log(1 + \exp(-y^{(i)} w^\top x^{(i)})).$$

As usual, our ERM $\hat{w} \in \mathbb{R}^d$ is what we get from minimizing $\hat{R}_n(w)$:

$$\hat{w} \in \arg\min_{w \in \mathbb{R}^d} \frac{1}{n}\sum_{i=1}^{n} \log(1 + \exp(-y^{(i)} w^\top x^{(i)})). \tag{1}$$

Once we have $\hat{w}$, we can make predictions by simply thresholding. On any example (x, y), we can predict $\hat{y} = 1$ if $\hat{w}^\top x > 0$ and $\hat{y} = -1$ if $\hat{w}^\top x < 0$. In this interpretation, $\hat{w}^\top x$ is a measure of "confidence," and we have a correct prediction (i.e. positive margin) if $w^\top x$ is the same sign as y.

Approach 2: Probabilistic Modeling. Suppose that instead of setting up an optimization problem (as in Approach 1), we want to take a probabilistic modeling approach from Week 7. In this approach, we still have the same input space $\mathcal{X} = \mathbb{R}^d$ but $\mathcal{Y} = \{0, 1\}$ instead of $\{-1, +1\}$. This is no problem — to convert from a dataset $D = \{(x^{(i)}, y^{(i)})\}_{i=1}^n$ with $\mathcal{Y} = \{-1, +1\}$ simply define

$$\tilde{y}^{(i)} = \begin{cases} 1 & \text{if } y^{(i)} = 1 \\ 0 & \text{if } y^{(i)} = -1 \end{cases}$$

and consider the dataset $D' = \{(x^{(i)}, \tilde{y}^{(i)})\}_{i=1}^n$ instead. We will use $\mathcal{A} = [0, 1]$ and we use the *logistic transfer function* $\sigma(z) = (1 + e^{-z})^{-1}$ (just the sigmoid function) atop a linear predictor, so we model the conditional probability $\Pr(y = 1 \mid x)$ as the following

$$\Pr(y = 1 \mid x) = \sigma(w^\top x) = \frac{1}{1 + \exp(-w^\top x)},$$

parameterized by $w \in \mathbb{R}^d$. With this assumption, we can maximize the likelihood to obtain $\hat{w}$ by solving

$$\hat{w} \in \arg\max_{w \in \mathbb{R}^d} \log \Pr(D'; w) = \sum_{i=1}^n \tilde{y}^{(i)} \log(\phi(w^\top x^{(i)})) + (1 - \tilde{y}^{(i)}) \log(1 - \phi(w^\top x^{(i)})),$$

recalling that $\tilde{y}^{(i)} \in \{0, 1\}$ and $\Pr(D'; w)$ is the probability of the dataset D' under the above parameterized distribution. Maximizing the likelihood is the same as minimizing the negative log-likelihood, i.e.

$$\hat{w} \in \arg\min_{w \in \mathbb{R}^d} -\left[\sum_{i=1}^n \tilde{y}^{(i)} \log(\phi(w^\top x^{(i)})) + (1 - \tilde{y}^{(i)}) \log(1 - \phi(w^\top x^{(i)}))\right]. \tag{2}$$

Problem 1(a) (5 points)

Prove that minimizing the negative log-likelihood under the logistic transfer function assumption above is *equivalent* to minimizing the ERM under the logistic loss function. That is, prove that the $\hat{w}$ obtained in Equation (1) is the same as the $\hat{w}$ obtained in Equation (2). *Hint: It may help to focus on each term of the sums, for each case of* $y^{(i)}$.

In either view, logistic regression is, at its heart, a binary classification method (that can be easily extended to multiple classes — see future labs/lectures). The simplest binary classification problems are those in which the data are *linearly separable*. Recall that a dataset $D = \{(x^{(i)}, y^{(i)})\}$ is linearly separable if there exists a $w \in \mathbb{R}^d$ and offset $w_0 \in \mathbb{R}$ such

that $w^\top x^{(i)} + w_0 > 0$ for every point $(x^{(i)}, y^{(i)})$ such that $y^{(i)} = +1$ and $w^\top x^{(i)} + w_0 < 0$ for every point $(x^{(i)}, y^{(i)})$ such that $y^{(i)} = -1$.

Problem 1(b) (2 points)

For simplicity, consider the case of logistic regression without a bias term w_0. Show that the decision boundary of logistic regression is given by $\{x \in \mathbb{R}^d : w^\top x = 0\} \subseteq \mathbb{R}^d$.

When working with the sigmoid function and log-likelihood, it is often useful to use the following facts, which you can use without proof in the following (though they're easy to prove – you can try yourself!):

- Derivative of sigmoid: $\frac{d}{dz}\sigma(z) = \sigma(z)(1 - \sigma(z))$.
- Derivative of log-sigmoid: $\frac{d}{dz}\log(\sigma(z)) = 1 - \sigma(z)$.
- Derivative of log-sigmoid: $\frac{d}{dz}\log(1 - \sigma(z)) = -\sigma(z)$.

Problem 1(c) (3 points)

Suppose that the data are linearly separable and, by using gradient descent, 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 to $\hat{w}$. This means that MLE is not well-defined in this case. *Hint: To show this, let $L(w) = \log \Pr(D'; w)$ be the log-likelihood (defined above). Then, as a function of c, consider $g(c) = L(c\hat{w})$, and show that the derivative $g'(c) > 0$. This shows that making c larger always makes g larger.*

As shown in Problem 1(c), 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* the data. In this part, we will apply regularization to fix this problem. For $\lambda > 0$, the ℓ_2-regularized logistic regression objective is defined as:

$$F_{\log}(w) := \frac{1}{n}\sum_{i=1}^{n} \log\left(1 + \exp\left(-y^{(i)} w^\top x^{(i)}\right)\right) + \lambda \|w\|_2^2.$$

Problem 1(d) (5 points)

Prove that the objective function $F_{\log}(w)$ is convex. You may use any of the facts mentioned in the lecture about convex optimization or any facts in the provided notes in Week 4 without proof.
Hint: It may be helpful to use some fact in the lecture notes about the convexity of norms for the second term. For the first term, can you analyze the Hessian?

We will now implement regularized logistic regression on a toy dataset using the skeleton code from `logreg_skeleton.py`.

Problem 1(e) (3 points)

Complete the `f_objective` function in the skeleton code, which computes the objective function $F_{\log}(w)$ for a given $w \in \mathbb{R}^d$ and $\lambda \geq 0$. Note that you may get numerical overflow when computing the exponential literally, e.g. try e^{1000} in `numpy`. You should use the `numpy` function `logaddexp` to get accurate calculations and prevent overflow (this implements the log-sum-exp trick).
Submission: Submit only the completed Python code for `f_objective`.

Problem 1(f) (2 points)

Complete the `fit_logistic_reg` function in the skeleton code by using the `minimize` function from `scipy.optimize`. Under the hood, this calls an optimization algorithm that you can think of as just an "upgraded gradient descent" (we won't reimplement GD on logistic regression on this problem set).
Submission: Submit only the completed Python code for `fit_logistic_reg`.

Problem 1(g) (5 points)

Using `fit_logistic_reg`, train several models on the data provided in `X_train.txt` and `y_train.txt` for different values of λ. Find the value of λ that minimizes the objective function on the validation set in `X_val.txt` and `y_val.txt` (an approximate answer is fine; we are just looking for the correct order of magnitude). Plot the objective function value as a function of λ (i.e. a plot with λ on the x-axis and $F_{\log}(w)$ on the y-axis).
Make sure you take the appropriate preprocessing steps, such as standardizing the data and adding a column for the bias term. You may use code from Problem Set 2 to do this.
Submission: Submit only your plot for this problem and state the value of λ you found does well on the validation data.

Problem 2: SVM with Pegasos Algorithm (50 points)

In lecture, we learned that SVM could be solved using standard convex optimization techniques in either primal or dual form. In this section, we explore an alternative method for solving the SVM problem known as the Pegasos algorithm. To understand the Pegasos algorithm, we introduce the concept of *subgradients* and *subgradient descent*, which generalizes gradient descent to convex functions that might be nondifferentiable at some points.

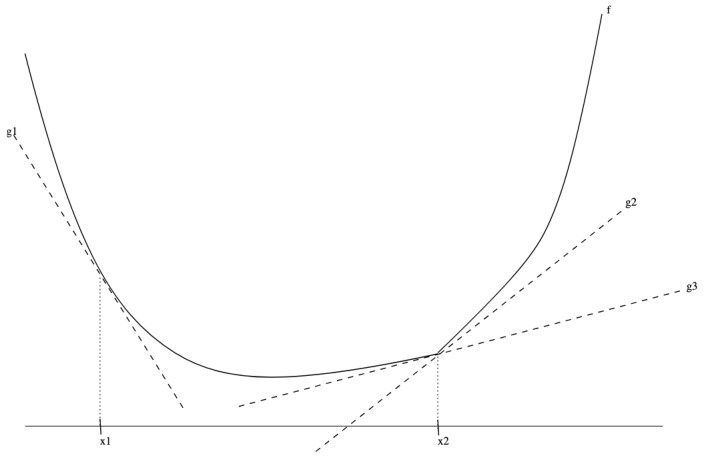
Subgradients. Recall that a differentiable function $f : \mathbb{R}^d \to \mathbb{R}$ is convex by the *first-order condition* for convexity if:

$$f(y) \geq f(x) + \nabla f(x)^\top (y - x),$$

for all $x, y \in \text{dom} f$. The definition of *subgradient* generalizes this notion to functions that may not have derivatives at some points. We say that a vector $g \in \mathbb{R}^d$ is a *subgradient* of $f : \mathbb{R}^d \to \mathbb{R}$ at $x \in \text{dom} f$ if for all $y \in \text{dom} f$,

$$f(y) \geq f(x) + g^\top (y - x).$$

Notice that if f is convex *and* differentiable, then its gradient at x is also a subgradient. However, functions that are *nondifferentiable* can also have subgradients. In fact, at a point x, there can be many subgradients. For example, see the following figure, taken from Boyd et al.'s class notes on subgradients where the convex function is differentiable at x_1 but not at x_2.



A function f is called *subdifferentiable* at x if there exists at least one subgradient at x. The set of subgradients of f at x is called the *subdifferential* of f at x, and is denoted $\partial f(x)$. There may be 0, 1, or infinitely many subgradients at any point.

As an example, take the absolute value function $f(x) = |x|$. For $x < 0$, the subgradient is unique, and it is just the gradient at x: $\partial f(x) = \{-1\}$. Similarly, for $x > 0$, we have

$\partial f(x) = \{1\}$. At $x = 0$, checking the definition involves checking which g satisfy:

$$|y| \geq |x| + g \cdot (y - x) \iff |y| \geq gy \quad \text{at } x = 0.$$

Solving for g, we get that $g \in [-1, 1]$ all satisfy the inequality, so our subgradient at $x = 0$ is the set $\partial f(0) = [-1, 1]$.

Below, let us derive a couple of properties that make it easier to work with the hinge loss.

Problem 2(a) (5 points)

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

Problem 2(b) (3 points)

Using Problem 2(a), give a subgradient of the hinge loss for a linear hypothesis:

$$g(w) = \max\{0, 1 - yw^\top x\}.$$

In this question you will build an SVM using the Pegasos algorithm. To align with the notation used in the Pegasos paper[1], we're considering the following formulation of the SVM objective function:

$$\min_{w \in \mathbb{R}^d} \frac{\lambda}{2}\|w\|^2 + \frac{1}{n}\sum_{i=1}^{n} \max\left\{0, 1 - y^{(i)} w^\top x^{(i)}\right\}.$$

Note that, for simplicity, we are leaving off the unregularized bias term w_0. Pegasos is stochastic *subgradient* descent using a step size rule $\eta_t = 1/(\lambda t)$.

For the purposes of this problem, we only need to understand that *subgradient descent* is simply a generalization of gradient descent where we replace the gradient with a subgradient at the current point. That is, the update rule is:

$$w_{t+1} \leftarrow w_t - \eta g^{(t)}$$

where $g^{(t)} \in \partial F(w_t)$ is *any* subgradient at w_t. The pseudocode of Pegasos is given below:

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

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

Problem 2(c) (5 points)

Consider the SVM objective function for a single training point:

$$F_i(w) = \frac{\lambda}{2}\|w\|^2 + \max\left\{0, 1 - y^{(i)} w^\top x^{(i)}\right\}.$$

The function $F_i(w)$ is not differentiable everywhere. Specify where the gradient of $F_i(w)$ is undefined and give an expression for $\nabla F_i(w)$ where it is defined.

Problem 2(d) (2 points)

Show that the subgradient of $F_i(w)$ is given by

$$g_w = \begin{cases} \lambda w - y^{(i)} x^{(i)} & \text{for } y^{(i)} w^\top x^{(i)} < 1 \\ \lambda w & \text{for } y^{(i)} w^\top x^{(i)} \geq 1 \end{cases}.$$

You may use the following facts without proof:

1. If $f_1, \ldots, f_n \colon \mathbb{R}^d \to \mathbb{R}$ are convex functions and $f = f_1 + \cdots + f_n$, then $\partial f(x) = \partial f_1(x) + \cdots + \partial f_n(x)$.
2. For $\alpha \geq 0$, $\partial(\alpha f)(x) = \alpha \partial f(x)$.

It may help to use Problem 2(a) and 2(b) for this.

You should convince yourself that if your step size rule is $\eta_t = 1/(\lambda t)$, then doing SGD with the subgradient direction from Problem 2(d) is the same as given in the pseudocode above.

Sentiment Classification with SVM. We will now turn to coding. You will be using SVM to predict whether the sentiment of a movie review is *positive* or *negative*. We represent each review by a vector $x \in \mathbb{R}^d$ where d is the size of the word dictionary and x_i is equal to hte number of occurrences of the ith word in review x. The corresponding labels are $y = 1$ for a positive review and $y = -1$ for a negative review.

Our dataset will be 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 `svm_skeleton.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}`. In `svm_skeleton.py`, we have also provided a class `reviewInstance` that converts lists of words into this sparse dictionary representation.

We will be using linear classifiers of the form $f(x) = w^\top x$, and we can store the 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 w and 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, `svm_skeleton.py` includes two functions for working with sparse vectors: a dot product between two vectors represented as dictionaries (in `dotProduct`) and a function that increments one sparse vector by a scaled multiple of another vector (in `increment`), which is a very common operation.

Problem 2(e) (2 points)

Implement `pegasos_sgd_loss`, which should take in a `dict` for a particular review x, the label y for the review, a `dict` representing w, and regularization parameter $\lambda > 0$. It should output a scalar value, the loss of a term in the objective

$$\frac{\lambda}{2}\|w\|^2 + \max\left\{0, 1 - y^{(i)} w^\top x^{(i)}\right\}.$$

Hint: It'll be helpful to use the ***`dotProduct`*** *function provided.*
Submission: Submit the Python code for `pegasos_sgd_loss` only.

Problem 2(f) (3 points)

Implement `pegasos_sgd_gradient`, which should take in a `dict` for a particular review x, the label y for the review, a `dict` representing w, and regularization parameter $\lambda > 0$. It should output a dictionary representing the gradient of a term in the objective. *Hint: It'll be helpful to use the* `increment` *function provided..*
Submission: Submit the Python code for `pegasos_sgd_gradient` only.

Now, we will implement the Pegasos algorithm to run on a sparse data representation. The output should be a sparse weight vector w. A couple of things to note:

- Our Pegasos algorithm starts at $w = 0$, which corresponds to an empty dictionary.
- 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.
- 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.

Problem 2(g) (5 points)

Implement `pegasos`, following the Pegasos algorithm presented above. The function operates on `review_list`, which is a list of `reviewInstance` (this is your training dataset). It also takes in `watch_list` which is an argument meant for your validation dataset. The termination criterion will simply be if we reached the maximum number of epochs for a pre-determined number of steps. We have included a gradient checker in `gradient_checker_for_pegasos` for you to plug in with `pegasos`.
Submission: Submit the Python code for `pegasos` only.

If you run the algorithm you implemented in Problem 2(g), you may find that it's particularly slow because it works on dictionaries. We will re-implement it with the following trick.

Note that in every step of the Pegasos algorithm, we rescale every entry of w_t by the factor $(1-\eta_t\lambda)$. 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 $s_{t+1} = (1 - \eta_t\lambda)s_t$.

Problem 2(h) (5 points)

If the update is $w_{t+1} = (1 - \eta_t \lambda) w_t + \eta_t y^{(i)} x^{(i)}$, show that the Pegasos update step is equivalent to first computing

$$s_{t+1} = (1 - \eta_t \lambda) s_t$$

and then applying

$$w'_{t+1} = w'_t + \frac{1}{s_{t+1}} \eta_t y^{(i)} x^{(i)}.$$

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 will divide by zero in the calculation for $w'_{t=1}$. This only happens when $\eta_t = 1/\lambda$ (when $t = 1$ with our step size rule). 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 $s_{t+1} = 0$ and $w' = 0$. Thus, if we ever get $s_{t+1} = 0$, we can simply set it back to 1 and reset w'_{t+1} to 0 which is an empty dictionary in a sparse representation.

Problem 2(i) (5 points)

Use the trick in Problem 2(h) to implement `pegasos_fast`, which runs the Pegasos algorithm with the same parameters as `pegasos` in Problem 2(g), but uses this computation trick.
Submission: Submit only your code for `pegasos_fast`.

At this point, you should check for yourself that your implementations of `pegasos` and `pegasos_fast` both result in essentially the same weight vector.

The ultimate quantity in binary classification that we want to make small is the *classification error*, which is simply the risk under the zero-one loss.

$$R(h) = \mathbb{E}[\ell_{01}(h(x), y)] = \mathbb{E}[\mathbf{1}\{h(x) \neq y\}] = \Pr[h(x) \neq y].$$

Of course, without knowing the underlying data distribution, we can only ever get our hands on the classification loss on a sample (the *empirical risk* under zero-one loss):

$$\hat{R}_n(h) = \sum_{i=1}^{n} \mathbf{1}\left\{h(x^{(i)}) \neq y^{(i)}\right\}.$$

Also recall that to make a binary prediction with the SVM, we *threshold* the score $w^\top x$. Typically, we want to predict $+1$ if $w^\top x > 0$ and -1 if $w^\top x < 0$. What we do when $w^\top x = 0$ is arbitrary (you can set this to whichever prediction you like).

Problem 2(j) (5 points)

Implement `classification_error` and `svm_predict`. The function `svm_predict` should simply take a single review x predict 1 or -1 based on the sign of $w^\top x$. The function `classification_error` should take in a list of labeled reviews (i.e. a dataset of $(x^{(i)}, y^{(i)})$ pairs) and output the classification loss (zero-one loss) on the sample.
Submission: Submit your code for both `classification_error` and `svm_predict`.

Finally, we will use `pegasos_fast` to minimize our error on our held-out test set. For this part of the problem, you will experiment yourself with the regularization parameter and look for one that has acceptable performance on the test set.

Problem 2(k) (5 points)

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 λ.
Hint: The error you get with the best regularization parameter should get you error closer around 17-18%. If not, you should train for longer.
Submission: Submit only your plot and state the value of λ that gave you acceptable error.

Finally, applications with natural language are particularly nice in that one can often interpret why a model performed well or poorly on a specific examples, and it is sometimes not very difficult to come up with better features to fix a problem. The first step can be to take a closer look at the errors a model makes.

The next problem involves some open ended experimentation.

Problem 2(l) (5 points)

Choose any example $x = (x_1, \ldots, x_d) \in \mathbb{R}^d$ (corresponding to a review) that the model got wrong. We want to investigate which feature (word) contributed to the incorrect prediction. One way to rank the importance of the features is to sort by the size of their contributions to the overall score. For your best $w \in \mathbb{R}^d$ you found in Problem 2(k), compute $|w_i x_i|$ where w_i is the weight of the ith feature in the prediction function weights, and x_i is the value of the ith feature in x. Create a table of the 20 most important features, sorted by $|w_i x_i|$, including the feature name, the feature value x_i, the feature weight w_i, and the product $w_i x_i$. In a few sentences, intuitively try to explain why the model was incorrect (using your own understanding of natural language).
Submission: A ranked table of the 20 features with the information above.

Problem 3: Kernel Methods (25 points)

In this problem, we will review and implement a few basic ideas from the lecture on kernels. This section is just exposition, but we recommend reading it to refresh and see everything from the lecture in one place.

Representer Theorem. Recall from lecture that, kernelization works nicely with optimization problems of the form

$$\min_{w \in \mathbb{R}^d} \; L\left(\langle w, x^{(1)} \rangle, \ldots, \langle w, x^{(n)} \rangle\right) + R\left(\sqrt{\langle w, w \rangle}\right),$$

where $w, x^{(1)}, \ldots, x^{(n)} \in \mathbb{R}^d$ and $\langle \cdot, \cdot \rangle$ is the standard inner product (dot product) on $\mathbb{R}^d$. The function $R : [0, \infty) \to \mathbb{R}$ is a nondecreasing regularization, while $L \colon \mathbb{R}^n \to \mathbb{R}$ is arbitrary (but in the context of the class, is empirical risk). Note that the outputs $y^{(i)}$ in the dataset are built in to the function already. We noted in lecture that this general form of function includes soft-margin SVM and ridge regression. Using the *representer theorem*, we showed that if the optimization problem has a solution, there is always a solution of the form $w = \sum_{i=1}^n \alpha_i x^{(i)}$, i.e. w is "in the span of the data."

Kernelized Problem. Let $\alpha \in \mathbb{R}^n$ be the vector of all the α_i's. Then, we can consider the "kernelized" optimization problem by plugging in the representation of w in terms of α into the original optimization problem:

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

where $K \in \mathbb{R}^{n \times n}$ is the *kernel matrix* (a.k.a. *Gram matrix*) defined by $K_{ij} = k(x^{(i)}, x^{(j)}) = \langle x^{(i)}, x^{(j)} \rangle$. We saw this explicit form for soft-margin SVM and ridge regression. On a new $x \in \mathbb{R}^d$, we can predict using

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

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

Kernel Trick (with explicit feature map). The *kernel trick* is to swap out occurrences of the kernel $k(\cdot, \cdot)$ (and the corresponding kernel matrix K) with another kernel. For example, we could replace $k(x^{(i)}, x^{(j)}) = \langle x^{(i)}, x^{(j)} \rangle$ by $k'(x^{(i)}, x^{(j)}) = \langle \psi(x^{(i)}), \psi(x^{(j)}) \rangle$ where $\psi \colon \mathbb{R}^d \to \mathbb{R}^D$ is any arbitrary feature mapping. In this case, the recovered $w \in \mathbb{R}^D$ can be written as $w = \sum_{i=1}^n \alpha_i \psi(x^{(i)})$ and predictions would be given by $\langle w, \psi(x^{(i)}) \rangle$. If $D \gg d$, then we have implicitly found a linear function in a higher dimensional space.

RBF Kernel. More interestingly, we can replace $k(\cdot, \cdot)$ by another kernel $k''(x^{(i)}, x^{(j)})$, for which we do not even know or cannot explicitly write down a corresponding feature map ψ. Our main example is the *radial basis function (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 explicitly since it would be infinite dimensional. Predictions must be done using $\alpha \in \mathbb{R}^n$, with $f(x) = \sum_{i=1}^{n} \alpha_i k(x^{(i)}, x)$.

First, we will get some familiarity with the definition of kernels on a few simple examples.

Problem 3(a) (6 points)

Consider an unlabeled dataset $D = \{x^{(1)}, x^{(2)}, x^{(3)}\} = \{-1, 0, 2\}$. Consider the following three symmetric functions on $\mathbb{R}$:

1. $k_A(x, z) = 1 + xz$
2. $k_B(x, z) = x^2 z^2$
3. $k_C(x, z) = 1 - |x - z|$.

For each of k_A, k_B, and k_C, form the 3×3 kernel matrix K with $K_{ij} = k(x^{(i)}, x^{(j)})$ and state the kernel matrix for each function. Then, for each function, state and prove whether each is a kernel or not with the definition of a kernel learned in lecture. You may use any of the definitions of PSD that you like; if you want, you can also compute eigenvalues using Python.

For the rest of this problem, we will be implementing some basic kernels. **NOTE:** Your implementation of kernel methods below should never make any reference to w or the feature map ψ! Your learning routine should return α, rather than w, and your prediction function should also use α rather than w. This will allow us to work with kernels that correspond to infinite-dimensional feature vectors.

There are many different families of kernels. So far, we have spoken about linear kernels, RBF kernels, and polynomial kernels. The last two kernel types have parameters (σ for the RBF kernel; offset parameter a and degree d for the polynomial kernel). For simplicity, throughout this problem, we will assume our input space is $\mathcal{X} = \mathbb{R}$. This allows us to represent a collection of n inputs as a matrix $X \in \mathbb{R}^{n \times 1}$. Refer to `kernel_skeleton.ipynb` for the skeleton code.

Problem 3(b) (6 points)

Write functions `RBF_kernel` and `polynomial_kernel` that compute the RBF kernel

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

and the polynomial kernel

$$k_{\text{poly}(a,d)}(x, x') = (a + \langle x, x' \rangle)^d.$$

The linear kernel $k_{\text{linear}}(x, x') = \langle x, x' \rangle$ has been done for you in the skeleton 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 matrix $M \in \mathbb{R}^{n_1 \times n_2}$ where $M_{ij} = k(W_i, X_j)$. That is, the (i, j) entry of M should be the 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 `cdis(X1, X2, 'sqeuclidean')` in the package `scipy.spatial.distance`.
Submission: Include the implemented Python code for the RBF kernel and the polynomial kernel.

Problem 3(c) (4 points)

Use the linear kernel function defined in the code (`linear_kernel`) to compute the kernel matrix on the set of unlabeled dataset $D_X = \{-4, -1, 0, 2\}$.
Hint: Not a trick question – this should be one or two lines.
Submission: Include *both* the code you use to compute the kernel matrix using `linear_kernel` and the output.

Suppose we have the labeled dataset $D_{X,Y} = \{(-4, 2), (-1, 0), (0, 3), (2, 5)\}$, where these ordered pairs are each $(x^{(i)}, y^{(i)})$ pairs for $x^{(i)} \in \mathbb{R}$ and $y^{(i)} \in \mathbb{R}$. Then, by the representer theorem, the final prediction function will be in the span of the functions $x \mapsto k(x_0, x)$ for $x_0 \in D_X$ (defined in Problem 3(c)). This set of functions looks quite different depending on the kernel function we use. The set of functions $x \mapsto k_{\text{linear}}(x_0, x))$ for $x_0 \in D_X$ has been provided for $x \in [-6, 6]$ for the linear kernel.

Problem 3(d) (4 points)

Plot the set of functions $x \mapsto k_{\text{poly}(1,3)}(x_0, x)$ for $x_0 \in D_X$ for $x \in [-6, 6]$, following the example of k_{linear}. Plot the set of functions $x \mapsto k_{\text{RBF}(1)}(x_0, x)$ for $x_0 \in D_X$ for $x \in [-6, 6]$, following the example of k_{linear}. Note that the parameter values for the polynomial kernel are $a = 1$ and $d = 3$ while the parameter value for the RBF kernel is $\sigma = 1$.
Submission: Two plots, one for the polynomial kernel and one for the RBF kernel.

Note: You may find it helpful to use partial application on your kernel functions. For ex-

ample, 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.

Problem 3(e) (5 points)

By the representer theorem, the final predictions 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 inputs in the training set. We will use the class `Kernel_Machine` in the skeleton code to make predictions with different kernels. Complete the `predict` function of the class `Kernel_Machine`. Construct a `Kernel_Machine` object with the RBF kernel with $\sigma = 1$ with prototype points at $-1, 0, 1$ and corresponding weights $\alpha = (1, -1, 1)$. Plot the resulting function.

Submission: Submit both your Python code that completes the `predict` function and the plot of your kernel machine the RBF kernel.