CPSC 440/550 Machine Learning (Jan-Apr 2024)
Assignment 1 – SOLUTIONS

1 Matrix Notation, Quadratics, Convexity, and Gradients [20 points]

Some of the notes on the course page might be useful to refresh some mathematical tools used in CPSC 340; in particular, your instance of 340 may not have covered positive semi-definite matrices, so check those notes out if needed. Each part is worth [2 points].

[1.1] Consider the function

$$f(x) = \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}x_i x_j + \sum_{i=1}^{n} b_i x_i + c,$$

where x is a vector of length n with elements x_i, b is a vector of length n with elements b_i, and A is an $n \times n$ matrix with elements a_{ij} (not necessarily symmetric). Write this function in matrix notation so that it uses A and b, and does not have summations or references to indices i.

Answer: $f(x) = x^T Ax + b^T x + c$.

[1.2] Write the gradient of f from the previous question, in matrix notation.

Answer: $\nabla f(x) = (A + A^T)x + b$.

[1.3] Show that f is convex if A is a symmetric, positive semi-definite matrix.

Answer: The Hessian is just $2A$, and $2A \succeq 0$ by assumption, showing convexity. A little more indirectly, you could also write $A = R^\mathsf{T} R$ and note that then $x^\mathsf{T} Ax = \|Rx\|^2$, and squared norms are convex, as are compositions of convex functions with affine transformations.

[1.4] When A is symmetric and *strictly* positive definite, give a linear system whose solution minimizes f in terms of x.

Answer: Equating the gradient with zero gives $2Ax = -b$, or $Ax = -\frac{1}{2}b$. Because f is convex, all minimizers satisfy this equation.

[1.5] Suppose that A is symmetric and only positive *semi*-definite. Will a "good" optimization algorithm, e.g. gradient descent with an appropriate step size schedule, necessarily find one of the solutions to your linear system from the last part? If not, where else could it go? *Hint: An example of a matrix that is psd but not strictly pd is $A = 0$.*

Answer: Not necessarily: with $A = 0$, the gradient is $-b$, which will never be zero – so many optimization algorithms will just diverge in the direction b forever, always improving the loss. This is because f is convex but not "coercive." In general, if there are any directions v with $Av = 0$ but $b^\mathsf{T} v < 0$, we can always improve by moving towards v.

[1.6] Consider weighted linear regression with an L2 regularizer, with regularization strength $1/\sigma^2$,

$$f(w) = \frac{1}{2}\sum_{i=1}^{n} v^{(i)}(y^{(i)} - w^\mathsf{T} x^{(i)})^2 + \frac{1}{2\sigma^2}\sum_{j=1}^{d} w_j^2,$$

where $\sigma > 0$ is finite and we have a vector v of length n containing the elements v^i. As usual, w is the weight vector, $x^{(i)}$ the ith input point, and $y^{(i)} \in \mathbb{R}$ its label. Write this function in matrix notation.

Note: you can use $\mathbf{X} \in \mathbb{R}^{n\times d}$ as the matrix with rows $(x^{(i)})^\mathsf{T}$, y as the vector containing the elements $y^{(i)}$, and V as a diagonal matrix containing the vector v along the diagonal.

Answer: One way to do this is

$$f(w) = \frac{1}{2}(Xw - y)^T V(Xw - y) + \frac{1}{2\sigma^2}\|w\|^2 .$$

You could alternately use $\|w\|^2 = w^T w$ or $(Xw - y)^T V(Xw - y) = \|Xw - y\|_V^2$.

[1.7] Assuming that $v^i \geq 0$ for all i, show that f from the previous part is convex. *(You can use Question [1.3] or not, as you prefer.)*

Answer: We can use that sums of convex functions are convex. The second term is a norm squared times a positive scalar. Squared norms are convex and multiplying by a non-negative scalar preserves convexity. The first term can be written as $g(Xw)$ for $g = \frac{1}{2}\sum_{i=1}^n v^i(y^i - w^T x^i)^2$. The function g is a sum of quadratic functions, each of which is convex since it has a second derivative of $v^i(x^i)^2 \geq 0$. Composing a convex g with the linear function Xw preserves convexity so the first term is convex.

Alternately, you could use that $\nabla^2 f(w) = X^\mathsf{T} V X + (1/\sigma^2)I$. This is positive definite, because $V = V^{\frac{1}{2}} V^{\frac{1}{2}}$ so that $X^\mathsf{T} V X = (V^{\frac{1}{2}} X)^\mathsf{T}(V^{\frac{1}{2}} X)$ is psd, and it's still psd if you add $\frac{1}{\sigma^2}I$ (increasing each eigenvalue by $\frac{1}{\sigma^2} > 0$).

[1.8] Assuming that we have $v^i \geq 0$ for all i, give a linear system whose solution minimizes f in terms of w. *(Again, use the previous results or not, your choice.)*

Answer: We have

$$\nabla f(w) = X^\mathsf{T} V(Xw - y) + \frac{1}{\sigma^2} w.$$

Equating this with zero we get

$$(X^\mathsf{T} V X + (1/\sigma^2)I)w = X^\mathsf{T} V y.$$

This f is guaranteed to be strictly convex, since the Hessian is $2VX + \frac{2}{\sigma^2}I \succ 0$, so this is a minimizer.

[1.9] If we fit a linear classifier with the exponential loss (used in older boosting algorithms, among other places), it would take the form

$$f(w) = \sum_{i=1}^n \exp(-y^{(i)} w^\mathsf{T} x^{(i)}).$$

Derive the gradient of this loss function.

Answer: In summation notation:

$$\nabla f(w) = \sum_{i=1}^n -y^{(i)} \exp(-y^{(i)} w^\mathsf{T} x^{(i)}) x^{(i)},$$

or in matrix notation as

$$\nabla f(w) = X^\mathsf{T} r,$$

where $r^{(i)} = -y^{(i)} \exp(-y^{(i)} w^\mathsf{T} x^{(i)})$.

[1.10] The support vector regression objective function is

$$f(w) = \sum_{i=1}^n \max\left\{0, \left|w^\mathsf{T} x^{(i)} - y^{(i)}\right| - \epsilon\right\} + \frac{\lambda}{2}\|w\|^2$$

where ϵ is a non-negative hyper-parameter. It is similar to the L1 robust regression loss, but "doesn't care" if your prediction is less than ϵ from the target (which can reduce overfitting). Show that this non-differentiable function is convex.

Answer: For the second term, squared norms are convex and $\lambda > 0$ means we are multiplying by a non-negative scalar which preserves convexity. We can re-write the terms in the first sum as $\max\{0, w^T x^i - y^i - \epsilon, -(w^T x^i - y^i) - \epsilon\}$ which is a maximum of linear functions. Linear functions are convex and max(convex) is convex and sum(convex) is convex so the first term is convex. So the function is a sum of convex functions and hence is convex.

2 Machine Learning Model Memory and Time Complexities [18 points]

Answer the following questions using big-O notation, and a brief explanation. Your answers may involve n, d, and perhaps additional quantities defined in the question. As an example, the (linear) least squares model has $\mathcal{O}(d)$ parameters, requires $\mathcal{O}(d)$ time for prediction, and requires $\mathcal{O}(nd^2 + d^3)$ time to train.[1]

Each part is worth [2 points].

[2.1] What is the training time for least squares (linear regression by solving the normal equations) with L2 regularization?

Answer: The solution is $w = (X^\mathsf{T}X + \lambda I)^{-1}X^\mathsf{T}y$. Computing the matrix-vector product $X^\mathsf{T}y$ costs $\mathcal{O}(nd)$, while computing the matrix $X^\mathsf{T}X$ costs $\mathcal{O}(nd^2)$. Solving the $d \times d$ linear system costs $\mathcal{O}(d^3)$, giving $O(nd^2 + d^3)$. (If you also consider re-formulating to work with the kernel matrix, this can be reduced to $\mathcal{O}(nd\min\{n, d\} + \min\{d, n\}^3)$.)

[2.2] What is the prediction cost for a trained linear model on t test examples?

Answer: The trained model uses $\tilde{y}^{(i)} = w^\mathsf{T}\tilde{x}^{(i)}$, which is an inner-product costing $\mathcal{O}(d)$. Doing this t times costs $\mathcal{O}(dt)$.

[2.3] What is the storage space required to make predictions with a linear regression model using Gaussian RBF features (with lengthscale, aka bandwidth, σ)?

Answer: We need to store all the training examples, which costs $\mathcal{O}(nd)$. (We also need the n regression weights, which costs an additional $\mathcal{O}(n)$ but doesn't increase the cost in $\mathcal{O}$ notation.)

[2.4] What is the prediction time for linear regression with Gaussian RBFs (with lengthscale, aka bandwidth, σ) as features on t test examples?

Answer: For each test example, we need to compute the distance to each training example. Each distance calculation costs $\mathcal{O}(d)$, and we do this n times giving $\mathcal{O}(nd)$. With these n features, it costs $\mathcal{O}(n)$ to compute the inner product with the regression weights. Repeating this for t examples gives $\mathcal{O}(ndt)$.

[2.5] What is the cost of evaluating the support vector regression objective function from Question [1.10]?

Answer: The second term costs $\mathcal{O}(d)$ due to the norm calculation. The first term involves the $w^\mathsf{T}x^{(i)}$ inner product which costs $\mathcal{O}(d)$, and sums over n examples so has a cost of $\mathcal{O}(nd)$. This gives a cost of $\mathcal{O}(nd)$.

[2.6] What is the cost of trying to minimize the exponential loss from Question [1.9] by running t iterations of gradient descent?

Answer: Each gradient evaluation involves the $\mathcal{O}(d)$ cost of computing an inner product for each of the n examples. Updating the weights based on the gradient costs $\mathcal{O}(d)$, so the cost per iteration is $\mathcal{O}(nd)$. Doing this for t iterations thus costs $\mathcal{O}(ndt)$.

[2.7] What is the cost of trying to minimize the exponential loss by running t iterations of stochastic gradient descent?

Answer: This is the same as the previous question without the sum over examples, reducing the cost to $\mathcal{O}(dt)$.

[2.8] What is the storage space required for the k-means clustering algorithm?

[1] In this course, we assume matrix operations have the "textbook" cost where the operations are implemented in a straightforward way, e.g. with `for` loops. For example, we'll assume that multiplying two $n \times n$ matrices or computing a matrix inverse simply costs $\mathcal{O}(n^3)$, rather than the $\mathcal{O}(n^\omega)$ where ω is more like 2.37, which while true is only relevant for extremely huge matrices; $\mathcal{O}(n^3)$ is closer to the behaviour observed for actual practical matrix sizes.

Answer: There are k means, each of which has size d, giving $\mathcal{O}(kd)$. (You could also argue that the storage is $\mathcal{O}(kd+n)$ if you consider the clusters of the training examples as part of the model.)

[2.9] What is the cost of clustering t examples using an already-trained k-means model?

Answer: It costs $\mathcal{O}(d)$ to compare one example to one mean, so $\mathcal{O}(kd)$ to compare it to all means. Finding the minimum among k numbers costs $\mathcal{O}(k)$. Repeating this t times gives $\mathcal{O}(tkd)$. Fancier data structures could allow for faster lookup, but would also increase storage space.

Many people got very low grades on variants of this question from past years. If you're not sure how to answer questions like this, get help!

3 MAP Estimation [12 points]

In 340, we showed that under the assumptions of a Gaussian likelihood and Gaussian prior,

$$y^{(i)} \sim \mathcal{N}(w^\top x^{(i)}, 1), \quad w_j \sim \mathcal{N}\left(0, \frac{1}{\lambda}\right),$$

the MAP estimate is equivalent to solving the L2-regularized least squares problem

$$f(w) = \frac{1}{2}\sum_{i=1}^{n}(w^\top x^{(i)} - y^{(i)})^2 + \frac{\lambda}{2}\sum_{j=1}^{d} w_j^2,$$

in the "loss plus regularizer" framework. For each of the alternate assumptions below, write it in the "loss plus regularizer" framework (simplifying as much as possible):

[3.1] [4 points] Gaussian likelihood with a separate variance $\nu^{(i)} > 0$ for each training example, and Laplace prior with a separate scale $1/\lambda_j > 0$ for each variable,

$$y^{(i)} \sim \mathcal{N}(w^\intercal x^{(i)}, \nu^{(i)}), \quad w_j \sim \mathcal{L}\left(0, \frac{1}{\lambda_j}\right)$$

where a Laplace distribution $\mathcal{L}(\mu, b)$ has density $\frac{1}{2b}\exp(-|x-\mu|/b)$.

Answer: You could write it in summation notation as

$$f(w) = \frac{1}{2}\sum_{i=1}^{n}\frac{1}{\nu^{(i)}}(w^\intercal x^{(i)} - y^{(i)})^2 + \sum_{j=1}^{d}\lambda_j|w_j|.$$

Or in matrix notation: define Θ as a diagonal matrix with the $1/\nu^{(i)}$ along the diagonal and similarly for Λ and the λ_j,

$$f(w) = \frac{1}{2}(Xw-y)^T\Theta(Xw-y) + \|\Lambda w\|_1.$$

or you could write it in terms of the covariance $\Sigma = \Theta^{-1}$ as

$$f(w) = \frac{1}{2}(Xw-y)^T\Sigma^{-1}(Xw-y) + \|\Lambda w\|_1.$$

or if you use quadratic norms you can get things like

$$f(w) = \frac{1}{2}\|Xw-y\|_\Theta^2 + \|\Lambda w\|_1.$$

[3.2] [4 points] Robust student-t likelihood and a uniform prior for w,

$$p(y^{(i)}|x^{(i)}, w) = \frac{1}{\sqrt{\nu}B\left(\frac{1}{2}, \frac{\nu}{2}\right)}\left(1 + \frac{(w^\intercal x^{(i)} - y^i)^2}{\nu}\right)^{-\frac{\nu+1}{2}}, \quad w \sim \text{Uniform}\left(\{w : \|w\| \le C\}\right),$$

where B is the "Beta" function, and the parameter ν is called the "degrees of freedom."[2] You can use V_d as the volume of the d-dimensional unit ball $\{w : \|w\| \le 1\}$. *Hint: Think carefully about what a uniform prior density means for the posterior density of w.*

Answer:

$$f(w) = \frac{\nu+1}{2}\sum_{i=1}^{n}\log\left(1 + \frac{(w^T x^i - y^i)^2}{\nu}\right)$$

or you could write the second term in matrix notation as $\frac{\lambda}{2}\|w-u\|^2$.

[2]This likelihood is more robust than the Laplace likelihood, but leads to a non-convex objective.

[3.3] [4 points] If $y^{(i)}$ represents counts, we can use a Poisson-distributed likelihood, and take a Gaussian prior on w centered at $u \in \mathbb{R}^d$:

$$p(y^{(i)}|x^{(i)}, w) = \frac{\exp(y^{(i)} w^\intercal x^{(i)}) \exp(-\exp(w^\top x^{(i)}))}{y^{(i)}!}, \quad w_j \sim \mathcal{N}\left(u_j, \frac{1}{\lambda}\right)$$

Answer:

$$f(w) = \sum_{i=1}^{n} \left(-y^{(i)} w^\intercal x^{(i)} + \exp(w^\intercal x^{(i)})\right) + \frac{\lambda}{2} \sum_{j=1}^{d} (w_j - u_j)^2,$$

If we wanted, we could define $v_i = \exp(w^\top x^{(i)})$ and write this in matrix notation as

$$f(w) = -y^\intercal X w + 1^\top v + \frac{\lambda}{2} \left\| w - u \right\|^2,$$

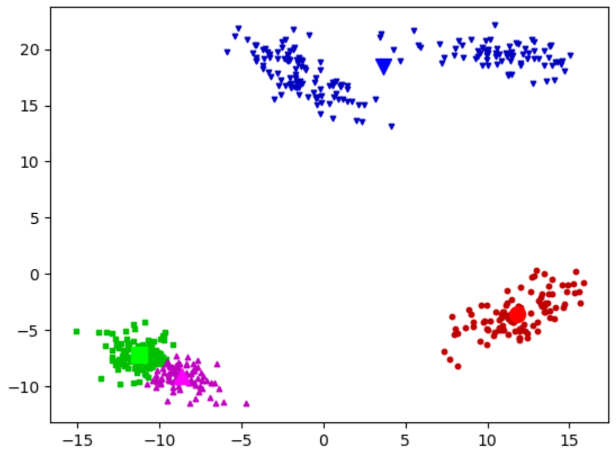
where 1 is a vector of ones.

4 K-Means Clustering [15 points]

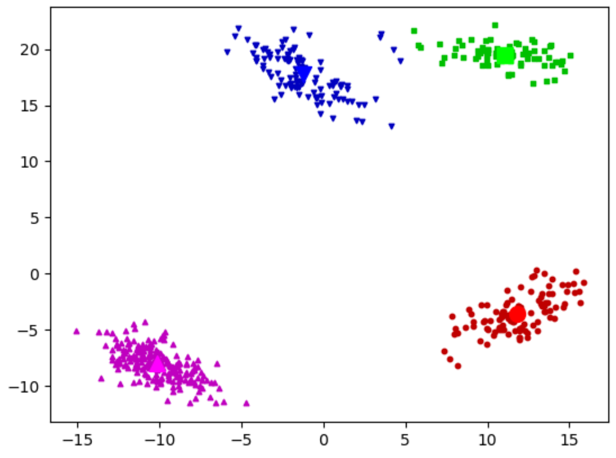
This and following questions use code available from the course webpage; get it from `a1.zip`.

You'll need Python 3 and the packages `numpy`, `scipy`, and `matplotlib`. If you don't have them installed already, install them with `conda install`, your system package manager, or `pip install numpy scipy matplotlib`.

If you run `main.py kmeans`, it will load a dataset with two features and a very obvious clustering structure. It will then apply the k-means algorithm with a random initialization. The result of applying the algorithm will thus depend on the randomization, but a typical run might look like this:



(Note that the colours are arbitrary due to the label switching problem.) But the "correct" clustering (that was used to make the data) is something more like this:



If you run the demo several times, it will find different clusterings. To select among clusterings for a *fixed* value of k, one strategy is to minimize the sum of squared distances between examples $x^{(i)}$ and their means $w_{y^{(i)}}$,

$$f(w^{(1)}, w^{(2)}, \dots, w^{(k)}, y^{(1)}, y^{(2)}, \dots, y^{(n)}) = \sum_{i=1}^{n} \left\| x^{(i)} - w^{(y^{(i)})} \right\|_2^2 = \sum_{i=1}^{n} \sum_{j=1}^{d} \left(x_j^{(i)} - w_j^{(y^{(i)})} \right)^2 .$$

where $y^{(i)} \in \arg\min_{c\in[n]} \left\| x^{(i)} - w^{(c)} \right\|$ is the index of the closest mean to x_i. This is a natural criterion because the steps of k-means alternately optimize this objective function in terms of the $w^{(c)}$ and the $y^{(i)}$ values.

[4.1] [3 points] Complete the function `KMeans.loss`, inside `k_means.py`, that takes in a data matrix $\mathbf{X}$, a vector of corresponding cluster assignments y, and a matrix of cluster means W, and computes this objective function. Hand in your code.

Answer: This might look something like

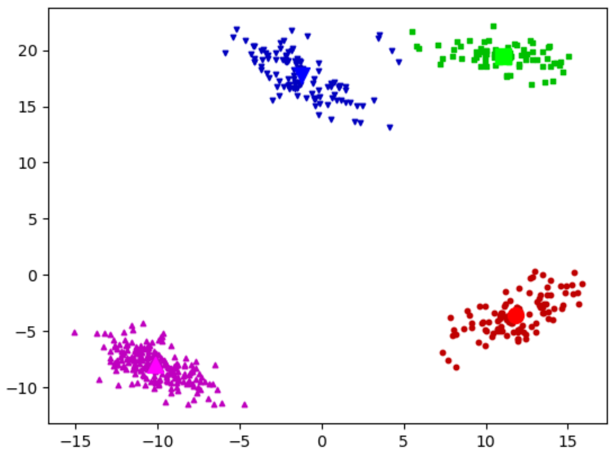
```
return np.sum((X - w[y]) ** 2)
```

[4.2] [1 points] Modify your code to, instead of/in addition to printing the number of labels that change on each iteration, print the value of `KMeans.loss` after each iteration. What trend do you observe? (No need to hand in code or specific loss values for this, just describe the trend.)

Answer: It decreases monotonically.

[4.3] [2 points] `main.py kmeans-best` will call the function `q_kmeans_best()` in `main.py`, which calls the `best_fit` function to rerun k-means 50 times and take the one with the lowest error. Complete the `best_fit` function, and hand in your code and the resulting plot.

Answer: This tends to give the true clustering, which looks like the figure above (up to a permutation of the colours).



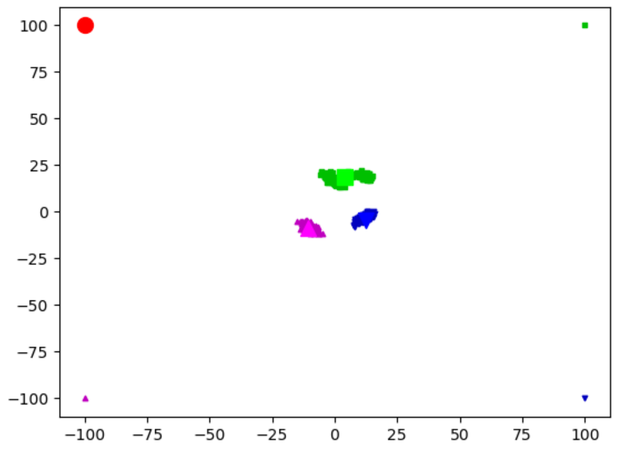
```
return min(
    (cls(X, 4, plot=False, log=False) for _ in range(50)),
    key=lambda model: model.loss(X),
)
```

[4.4] [3 points] Explain why the `KMeans.loss` function should not be used to choose k – even if you evaluate it on test data.

Answer: On training data, it will *always* simply pick the largest value of k, since adding a new prototype can only decrease the loss. Even on test data: since you have more clusters as k increases, the closest mean is still likely to be closer on new data with large values of k. Even on test data, this objective function would prefer the largest value of k.

[4.5] [1 points] The data in `cluster_data_2.npz` is exactly the same as `cluster_data.npz`, except that it has four outliers that are far away from the data. `main.py kmeans-outliers` will run your code from above to find the best of 50 runs on this data and save it as `figs/kmeans-outliers.png`. Hand in this plot; are you satisfied with it?

Answer: Although other variations are possible, this tends to to assign 3 clusters to the center points and then puts 1 cluster directly on one of the outliers, as in this plot:



This is annoying because k-means is wasting an entire cluster on a single outlier.

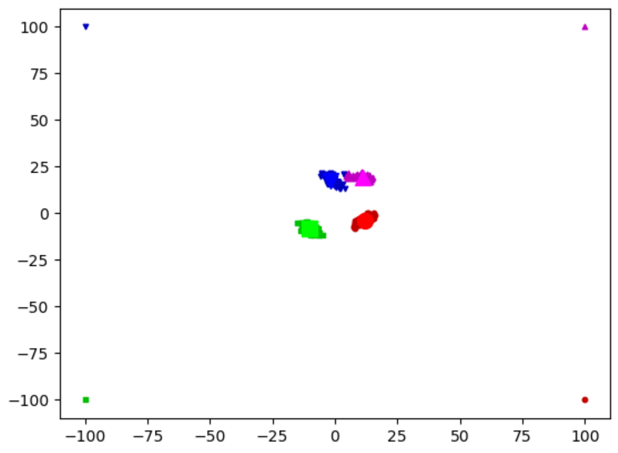
[4.6] [5 points] Implement the *k-medians* algorithm in `kmedians.py`, which assigns examples to the nearest $w^{(c)}$ in the L1-norm, and then updates all the $w^{(c)}$ by setting them to the "median" of the points assigned to the cluster (defining the d-dimensional median as the concatenation of the medians along each dimension). For this algorithm it makes sense to use the L1-norm version of the error (where $y^{(i)}$ now represents the closest centre in the L1-norm),

$$f(w^{(1)}, w^{(2)}, \ldots, w^{(k)}, y^{(1)}, y^{(2)}, \ldots, y^{(n)}) = \sum_{i=1}^{n} \left\| x^{(i)} - w^{(y^{(i)})} \right\|_1 = \sum_{i=1}^{n} \sum_{j=1}^{d} |x_j^{(i)} - w_j^{(y^{(i)})}|,$$

`main.py kmedians-outliers` will find the best of 50 models with $k = 4$ for you, once you've finished the `KMedians` class. Hand in your code and plot. Is this better?

Answer:

```python
# If you want, you could write this function to compute pairwise L1 distances
def l1_distances(X1, X2):
    # BEGIN SOLUTION
    n1, d = X1.shape
    n2, d2 = X2.shape
    assert d == d2

    return np.abs(X1[:, np.newaxis, :] - X2[np.newaxis, :, :]).sum(axis=2)
    # END SOLUTION


class KMedians(KMeans):
    # We can reuse most of the code structure from KMeans, rather than copy-pasting,
    # by just overriding these few methods. Object-orientation!

    def get_assignments(self, X):
        # BEGIN SOLUTION
        return np.argmin(l1_distances(X, self.w), axis=1)
        # END SOLUTION

    def update_means(self, X, y):
        # BEGIN SOLUTION
        for c in range(self.k):
            matching = y == c
            if matching.any():
                self.w[c] = np.median(X[matching], axis=0)
        # END SOLUTION

    def loss(self, X, y=None):
        w = self.w
        if y is None:
            y = self.get_assignments(X)

        # BEGIN SOLUTION
        return np.sum(np.abs(X - w[y]))
        # END SOLUTION
```



We get the four “true” clusters again, with the outliers assigned to the closest one!

5 Regularization and Hyper-Parameter Tuning [20 points]

If you run `main.py lsq`, it will:

1. Load a one-dimensional regression dataset.
2. Fit a least-squares linear regression model.
3. Draw a figure showing the training/testing data and what the model looks like.

Unfortunately, this is not a great model of the data, and the figure shows that a linear model with a y intercept of 0 is probably not suitable.

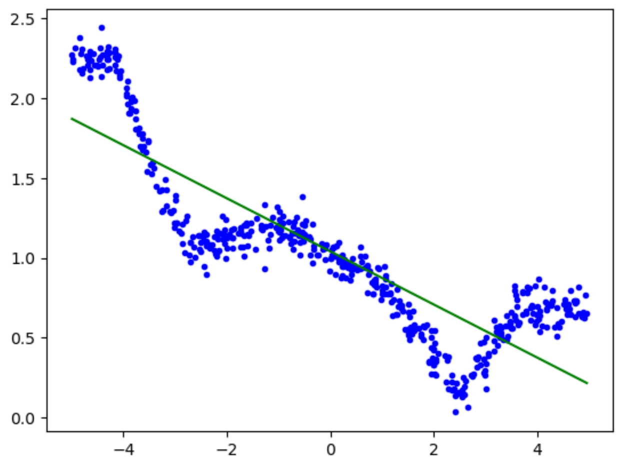
[5.1] [5 points] Finish the `LeastSquaresBias` class, in `least_squares.py`. Using this class should fit and predict with the model

$$y^{(i)} = w^{\mathsf{T}} x^{(i)} + b,$$

where both w and b are fit to data with least squares; implement it however you like (there are a few reasonable options). `main.py lsq-bias` will then run it for you. Hand in your new class and the updated plot.

Answer:

```python
class LeastSquaresBias:
    def __init__(self, X=None, y=None):
        if X is not None and y is not None:
            self.fit(X, y)

    # BEGIN SOLUTION rep:
    def featurize(self, X):
        bias_column = np.ones((X.shape[0], 1))
        return np.concatenate((bias_column, X), axis=1)

    # END SOLUTION

    def fit(self, X, y):
        # BEGIN SOLUTION
        self.base_model = LeastSquares(self.featurize(X), y)
        # END SOLUTION

    def predict(self, X):
        # BEGIN SOLUTION
        if self.base_model is None:
            raise RuntimeError("You must fit the model first!")

        return self.base_model.predict(self.featurize(X))
        # END SOLUTION
```



[5.2] [6 points] Allowing a non-zero y-intercept improves the prediction substantially, but the model still seems sub-optimal because there are obvious non-linear effects. Complete the model `leastSquaresRBFL2` that implements *least squares using Gaussian radial basis functions (RBFs) with L2-regularization.* (You can include an intercept or not, your choice.) `main.py lsq-rbf` will then run it for you.

Use `lam` for the L2 regularization parameter[3], and `sigma` for the lengthscale of the Gaussian features. Note that your L2 regularization should correspond to minimizing the loss function $\|\mathbf{X}w - y\|^2 + \lambda \|w\|^2$; some versions instead correspond to putting a $\frac{1}{n}$ in front of the loss term.

Hand in your function and the plot generated with $\lambda = 1$ and $\sigma = 1$.

Hint: The function `utils.euclidean_dist_squared`, which was also used in our k-means implementation, efficiently computes the squared Euclidean distance between all pairs of rows in two matrices.

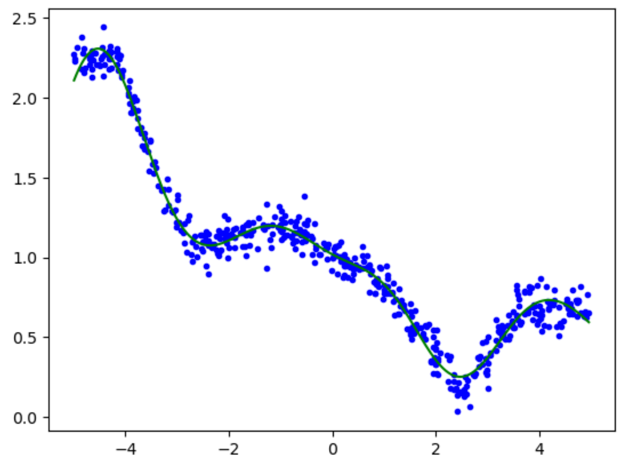
Answer:

```python
class LeastSquaresL2(LeastSquares):
    def __init__(self, X=None, y=None, lam=1):
        self.lam = lam
        super().__init__(X, y)

    def fit(self, X, y):
        n, d = X.shape
        self.w = np.linalg.solve(X.T @ X + self.lam * np.eye(d), X.T @ y)
```

```python
def gaussianRBF_feats(X, bases, sigma):
    # Not mandatory, but might be nicer to implement this separately.
    # BEGIN SOLUTION
    D2 = euclidean_dist_squared(X, bases)
    D2 /= -2 * (sigma**2)
    np.exp(D2, out=D2)
    return D2
    # END SOLUTION


class LeastSquaresRBFL2:
    def __init__(self, X=None, y=None, lam=1, sigma=1):
```

[3]We can't name it `lambda`, because that's a reserved word in Python. You could technically type λ, but there are a lot of Unicode characters for λ that all look really similar, so it gets annoying. . . .

```
        self.lam = lam
        self.sigma = sigma
        if X is not None and y is not None:
            self.fit(X, y)

    # BEGIN SOLUTION rep:
    def featurize(self, X):
        return gaussianRBF_feats(X, self.X, self.sigma)

    # END SOLUTION

    def fit(self, X, y):
        # BEGIN SOLUTION
        self.X = X
        self.base_model = LeastSquaresL2(self.featurize(X), y, lam=self.lam)
        # END SOLUTION

    def predict(self, X):
        # BEGIN SOLUTION
        if self.base_model is None:
            raise RuntimeError("You must fit the model first!")

        return self.base_model.predict(self.featurize(X))
        # END SOLUTION
```



[5.3] [6 points] Modify the script, in the function `lsq-rbf-split`, to split the training data into a "train" and "validation" set (you can use half the examples for training and half for validation), and use these to select λ and σ from some reasonable set of candidate values. You'll probably want to vary these by a few orders of magnitude either smaller or larger from 1.

(Although in real work you'd probably use some pre-coded helpers for this, code it yourself here.)

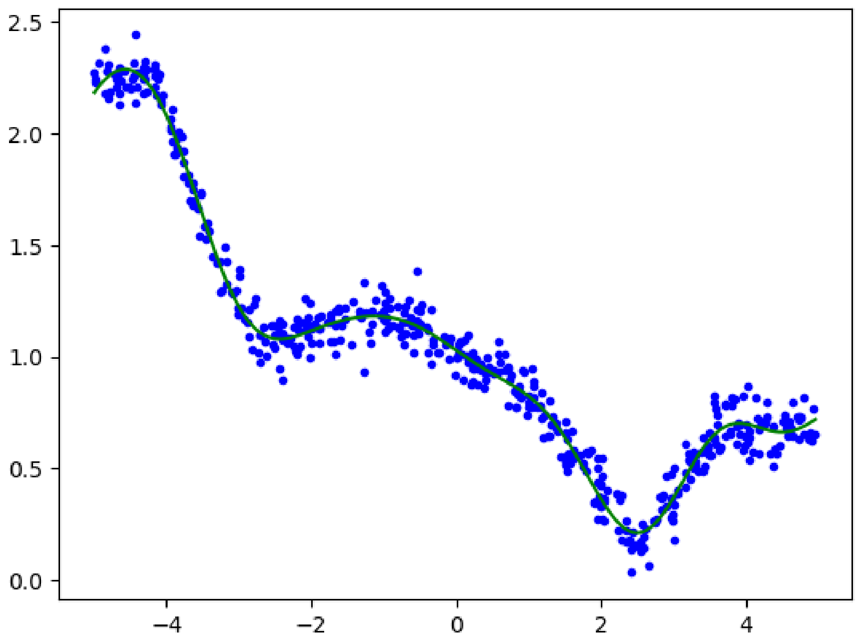
Hand in your modified function, the selected λ and σ, and the plot you obtain by refitting on the full dataset with the best values of λ and σ.

Answer:

```
def q_lsq_rbf_split():
```

```python
    X, y = load_dataset("basis_data", "X", "y")

    # BEGIN SOLUTION
    n, d = X.shape
    in_train = np.random.default_rng().choice(n, n // 2, replace=False)
    X_train = X[in_train]
    y_train = y[in_train]
    X_valid = X[~in_train]
    y_valid = y[~in_train]

    best_valid_error = np.inf
    best_lam = 1
    best_sigma = 1

    for lam in [0.001, 0.01, 0.1, 1, 10, 100, 1000]:
        for sigma in [0.001, 0.01, 0.1, 1, 10, 100, 1000]:
            model = LeastSquaresRBFL2(X_train, y_train, lam=lam, sigma=sigma)
            y_hat = model.predict(X_valid)
            valid_error = np.sum((y_hat - y_valid) ** 2)
            print(
                f"Validation error for lambda={lam:5}, sigma={sigma:5} is
                ↪  {valid_error:>7.2f}"
            )
            if valid_error < best_valid_error:
                best_valid_error = valid_error
                best_lam = lam
                best_sigma = sigma

    print(f"Best Validation Error: {best_valid_error:7.2f}")
```

```
Best Validation Error:    1.36
Best lambda: 0.001
Best sigma: 1
Training error = 0.0
```



[5.4] [3 points] Consider (but you don't have to implement) a model combining the first two parts of this question,

$$y^{(i)} = w^\top x^{(i)} + b + v^\top z^{(i)},$$

where $z^{(i)}$ are the Gaussian RBFs and v is a vector of regression weights for those features. Suppose that we first fit w and b assuming that $v = 0$ as in Question [5.1], and then we fit v with w and w_0 fixed (you could use your code for Question [5.3] to do this by modifying the $y^{(i)}$ values). Why would someone want to do this?

Hint: Think about how this model would behave on a test point $\tilde{x} = 10$.

Answer: This will probably not increase the accuracy within the range of the data, but it will keep the linear trend as you move away from the range of the data (the Gaussian RBFs go to zero and the linear model takes over). You might want this if you want to model the non-linear trend between the data points but think the linear trend will continue away from the data (so going to zero away from the data is the wrong thing to do).

6 Optimal matrix multiplication [15 points]

Suppose we have a bunch of matrices $M_1, M_2, \dots, M_k$, where M_i is of shape $d_{i-1} \times d_i$, and we want to find the $d_0 \times d_k$ matrix $M_1 M_2 \cdots M_k$. There are lots of ways to compute this, since matrix multiplication is associative: for example, if $k = 6$ we could do $M_1(M_2(M_3(M_4(M_5 M_6))))$, $M_1((M_2 M_3)((M_4 M_5) M_6))$, etc. The number of possibilities is exponential in k.

Recall that if A is $m \times n$ and B is $n \times p$, we can compute AB with $\mathcal{O}(mnp)$ scalar operations; this is typically a "good-enough" model for how much this costs in practice for the kinds of problems that appear in ML.

Fill in the `mmul_order` function in `mmul_order.py` to use dynamic programming to find the optimal order of multiplications.

Make sure you follow the comments there as to the input and output format of your function, and that your algorithm is $\mathcal{O}(k^3)$ or better.

`main.py mmul-order` will call the function for you with a few test cases; hand in your code and the cost and order for the final part.

Answer: The cost should be 11,893,035. If it's not that, then something's wrong (or my solution is broken. . . hopefully not!); look at their code a bit to see if something's obviously broken.

Note that if you compute the strings for *everything*, not just the path they actually use, that's probably going to be $\mathcal{O}(n^4)$ in the worst case! (Grader: check for this.)

The order can vary a little bit; my code printed out

```
Got order 1 x 2; 3 x 4; 3:4 x 5; 3:5 x 6; 3:6 x 7; 3:7 x 8; 3:8 x 9; 3:9 x 10;
          3:10 x 11; 3:11 x 12; 3:12 x 13; 3:13 x 14; 3:14 x 15; 3:15 x
          16; 1:2 x 3:16
```

but, for example, you could switch the first two since they don't depend on each other. The most natural other order for them to print would be

```
Got order 3 x 4; 3:4 x 5; 3:5 x 6; 3:6 x 7; 3:7 x 8; 3:8 x 9; 3:9 x 10; 3:10 x 11;
          3:11 x 12; 3:12 x 13; 3:13 x 14; 3:14 x 15; 3:15 x 16; 1 x 2;
          1:2 x 3:16
```

If it's neither of these, but they got the right cost, pay attention!

```python
def mmul_order(dims):
    # the first matrix is dims[0] by dims[1]; the last is dims[k-1] by dims[k]
    # so, matrix i (in *one*-indexing, as in the math) is dims[i-1] by dims[i]

    # you should specify your order as string specifying the order of multiplications:
    # for instance, right-to-left for a list of four matrices would be something like
    #    3 x 4; 2 x 3:4; 1 x 2:4
    # the _fmt() function above might help

    # feel free to either construct an explicit table or use functools.cache, which we
    ↪  imported for you

    # BEGIN SOLUTION

    # being a little wasteful: [0] indices and i >= j are useless
    # could also fill this in in a "bottom-up" order
    costs = np.zeros((len(dims), len(dims)), dtype=int)
    decisions = costs.copy()
    computed = np.zeros((len(dims), len(dims)), dtype=bool)

    def cost_for(i, j):
        # What's the most efficient way to multiply M[i] through M[j], inclusive?
        if computed[i, j]:
            return costs[i, j]
        if i == j:
            return 0
        if i > j:
            raise ValueError(f"something screwed up: {i=} {j=}")

        best_cost = float("inf")
        best_k = None
        # generally, for any i <= k < j,
        #   (    M[i] through M[k]    ) by ( M[k+1] through M[j] )
        #       dims[i-1] by dims[k]         dims[k] by dims[j]
        #   costing   cost(i, k) + dims[i-1] * dims[k] * dims[j] + cost(k+1, j)

        for k in range(i, j):
            cost = cost_for(i, k) + dims[i - 1] * dims[k] * dims[j] + cost_for(k + 1, j)
            if cost < best_cost:
                best_cost = cost
                best_k = k

        costs[i, j] = best_cost
        decisions[i, j] = best_k
        computed[i, j] = True
        return best_cost

    def order_for(i, j):
        if i == j:
            return []

        cost_for(i, j)  # make sure we've computed all the way down
        k = decisions[i, j]
        this_dec = [f"{_fmt(i, k)} x {_fmt(k+1, j)}"]
        return order_for(i, k) + order_for(k + 1, j) + this_dec

    optimal_cost = cost_for(1, len(dims) - 1)
    optimal_order = "; ".join(order_for(1, len(dims) - 1))
    # END SOLUTION

    return optimal_cost, optimal_order
```