CPSC 340 Assignment 5 – Due 2021-11-22

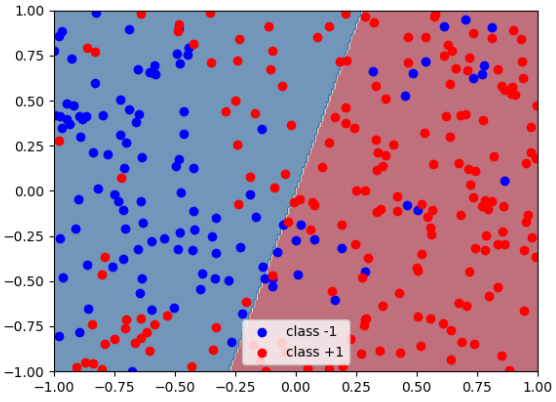
Pedram Amani - 73993008
Henry Xu - 40728164

Important: Submission Format [5 points]

Please make sure to follow the submission instructions posted on the course website. We will deduct marks if the submission format is incorrect, or if you're not using LaTeX and your submission is *at all* difficult to read – at least these 5 points, more for egregious issues. Compared to assignment 1, your name and student number are no longer necessary (though it's not a bad idea to include them just in case, especially if you're doing the assignment with a partner).

1 Kernel Logistic Regression [22 points]

If you run `python main.py -q 1` it will load a synthetic 2D data set, split it into train/validation sets, and then perform regular logistic regression and kernel logistic regression (both without an intercept term, for simplicity). You'll observe that the error values and plots generated look the same, since the kernel being used is the linear kernel (i.e., the kernel corresponding to no change of basis). Here's one of the two identical plots:



1.1 Implementing kernels [8 points]

Inside `kernels.py`, you will see classes named `PolynomialKernel` and `GaussianRBFKernel`, whose `__call__` methods are yet to be written. Implement the polynomial kernel and the RBF kernel for logistic regression. Report your training/validation errors and submit the plots from `utils.plot_classifier` for each case. You should use the kernel hyperparameters $p = 2$ and $\sigma = 0.5$ respectively, and $\lambda = 0.01$ for the regularization strength. For the Gaussian kernel, please do *not* use a $1/\sqrt{2\pi\sigma^2}$ multiplier.

Answer:

- Polynomial kernel

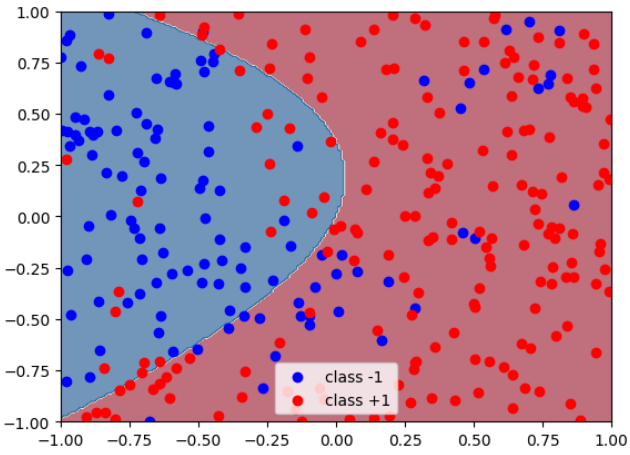
  Training error: 18.3%

  Validation error: 17.0%

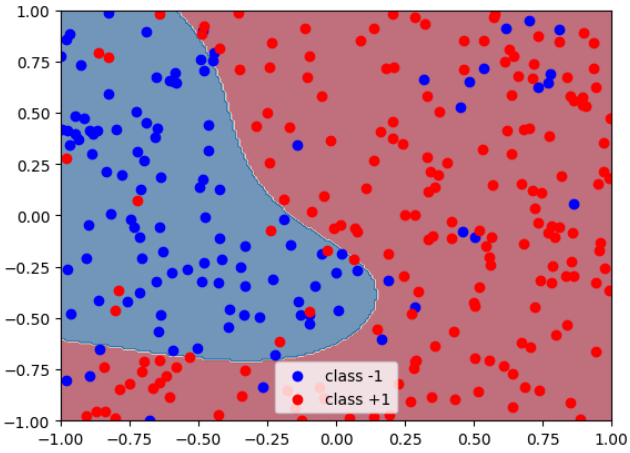
- Gaussian RBF kernel

  Training error: 12.7%

  Validation error: 11.0%



Polynomial Kernel



Gaussian RBF Kernel

1.2 Hyperparameter search [10 points]

For the RBF kernel logistic regression, consider the hyperparameter values $\sigma = 10^m$ for $m = -2, -1, \ldots, 2$ and $\lambda = 10^m$ for $m = -4, -3, \ldots, 2$. The function `q1_2()` has a little bit in it already to help set up to run a grid search over the possible combination of these parameter values. You'll need to fill in the `train_errs` and `val_errs` arrays with the results on the given training and validation sets, respectively; then the code

already in the function will produce a plot of the error grids. Submit this plot. Also, for each of the training and testing errors, pick the best (or one of the best, if there's a tie) hyperparameters for that error metric, and report the parameter values and the corresponding error, as well as a plot of the decision boundaries (plotting only the training set). While you're at it, submit your code. To recap, for this question you should be submitting: two decision boundary plots, the values of two hyperparameter pairs with corresponding errors, and your code.

Note: on the real job you might choose to use a tool like scikit-learn's `GridSearchCV` to implement the grid search, but here we are asking you to implement it yourself, by looping over the hyperparameter values.

Answer:

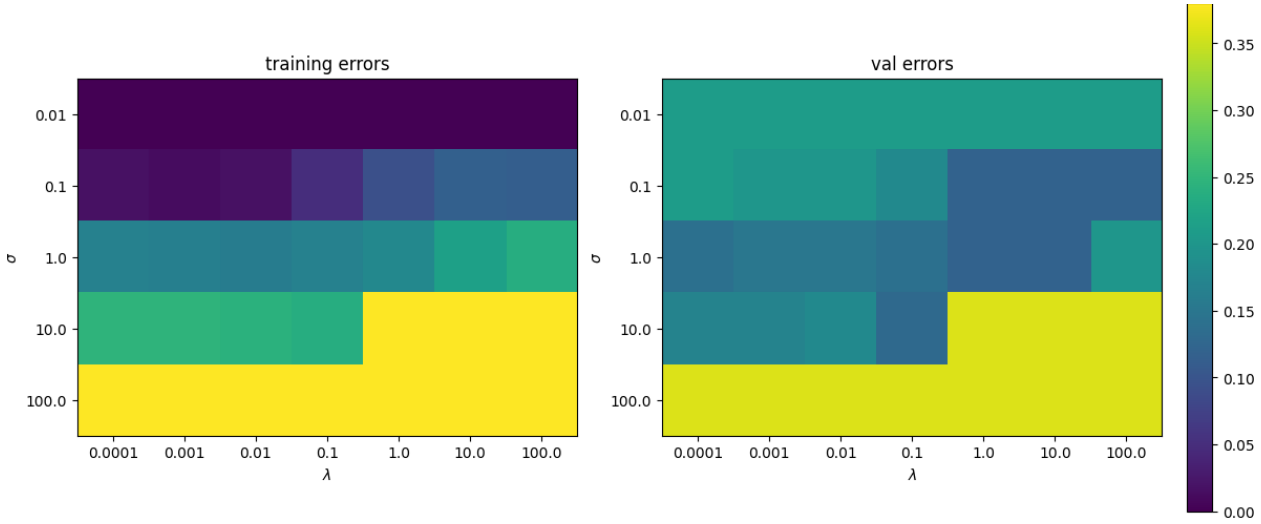
- Training error: 0 for $\sigma = 0.01, \lambda = 1$
- Validation error: 0.12% for $\sigma = 0.1, \lambda = 1$

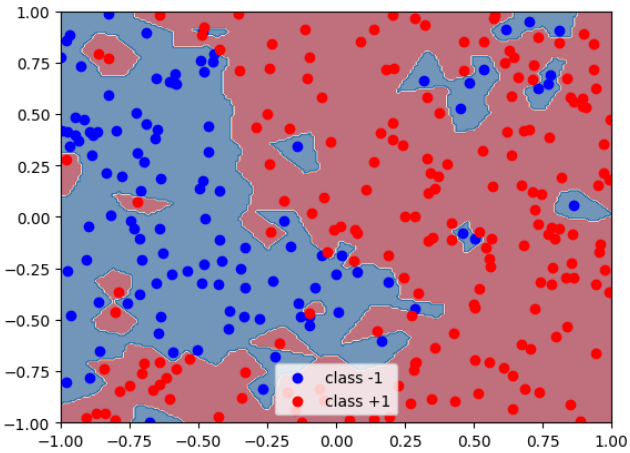
```
for i, sigma in enumerate(sigmas):
    for j, lammy in enumerate(lammys):
        loss_fn = KernelLogisticRegressionLossL2(lammy)
        optimizer = GradientDescentLineSearch()
        kernel = GaussianRBFKernel(sigma)
        klr_model = KernelClassifier(loss_fn, optimizer, kernel)
        klr_model.fit(X_train, y_train)

        train_errs[i, j] = np.mean(klr_model.predict(X_train) != y_train)
        val_errs[i, j] = np.mean(klr_model.predict(X_val) != y_val)
```

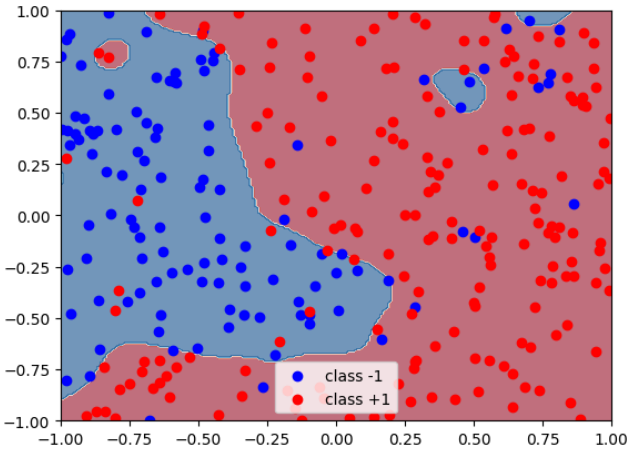
```
for i, (sigma, lammy) in enumerate([(0.01, 1.), (1., 0.01)]):
    loss_fn = KernelLogisticRegressionLossL2(lammy)
    optimizer = GradientDescentLineSearch()
    kernel = GaussianRBFKernel(sigma)
    klr_model = KernelClassifier(loss_fn, optimizer, kernel)
    klr_model.fit(X_train, y_train)

    fig = utils.plot_classifier(klr_model, X_train, y_train)
    utils.savefig(f"logRegRBF_boundary{i}.png", fig)
```





Using hyper-parameters that minimize training error



Using hyper-parameters that minimize validation error

1.3 Reflection [4 points]

Briefly discuss the best hyperparameters you found in the previous part, and their associated plots. Was the training error minimized by the values you expected, given the ways that σ and λ affect the fundamental tradeoff?

Answer: Lower values of σ and λ correspond to a higher model complexity. We expect the training error to decrease with increasing model complexity (i.e. more overfitting); and indeed we see that the training error decreases moving toward the upper-left corner of the colormap (lower σ and λ). A model with low complexity performs poorly in both the training and validation datasets.

To minimize the validation error, we need enough granularity (low enough σ) to capture the variation in our data but not so low as to overfit the training data. For this dataset, $\sigma \approx 0.1$ strikes a good balance.

2 MAP Estimation [16 points]

In class, we considered MAP estimation in a regression model where we assumed that:

- The likelihood $p(y_i \mid x_i, w)$ comes from a normal density with a mean of $w^T x_i$ and a variance of 1.
- The prior for each variable j, $p(w_j)$, is a normal distribution with a mean of zero and a variance of λ^{-1}.

Under these assumptions, we showed that this leads to the standard L2-regularized least squares objective function,

$$f(w) = \frac{1}{2}\|Xw - y\|^2 + \frac{\lambda}{2}\|w\|^2,$$

which is the negative log likelihood (NLL) under these assumptions (ignoring an irrelevant constant). For each of the alternate assumptions below, show the corresponding loss function [each 4 points]. Simplify your answer as much as possible, including possibly dropping additive constants.

1. We use a Gaussian likelihood where each datapoint has its own variance σ_i^2, and a zero-mean Laplace prior with a variance of λ^{-1}.

$$p(y_i \mid x_i, w) = \frac{1}{\sqrt{2\sigma_i^2\pi}} \exp\left(-\frac{(w^T x_i - y_i)^2}{2\sigma_i^2}\right), \quad p(w_j) = \frac{\lambda}{2}\exp(-\lambda|w_j|).$$

   You can use Σ as a diagonal matrix that has the values σ_i^2 along the diagonal.

   Answer:

$$\begin{aligned} f(w) &= -\log(p(X, w \mid y)) = -\log(p(y \mid X, w)) \cdot p(w)) \\ &= -\log\left(\prod_i p(y_i \mid x_i, w)\cdot\right) - \log\left(\prod_j p(w_j)\right) \\ &= -\sum_i \left(-\frac{(w^T x_i - y_i)^2}{2\sigma_i^2} - \log\left(\sqrt{2\sigma_i^2\pi}\right)\right) - \sum_j \left(\log(\frac{\lambda}{2}) - \lambda|w_j|\right) \\ &\sim \frac{1}{2}(Xw - y)^T\Sigma^{-1}(Xw - y) + \lambda\|w\|_1 \end{aligned}$$

2. We use a Laplace likelihood with a mean of $w^T x_i$ and a variance of 8, and we use a zero-mean Gaussian prior with a variance of σ^2:

$$p(y_i \mid x_i, w) = \frac{1}{4}\exp\left(-\frac{1}{2}|w^T x_i - y_i|\right), \quad p(w_j) = \frac{1}{\sqrt{2\pi}\,\sigma}\exp\left(-\frac{w_j^2}{2\sigma^2}\right).$$

   Answer:

$$\begin{aligned} f(w) &= -\log(p(X, w \mid y)) = -\log(p(y \mid X, w)) \cdot p(w)) \\ &= -\log\left(\prod_i p(y_i \mid x_i, w)\cdot\right) - \log\left(\prod_j p(w_j)\right) \\ &= -\sum_i \left(\log\left(\frac{1}{4}\right) - \frac{1}{2}|w^T x_i - y_i|\right) - \sum_j \left(-\log(\sqrt{2\pi}\sigma) - \frac{w_j^2}{2\sigma^2}\right) \\ &\sim \frac{1}{2}\|(Xw - y)\|_1 + \frac{1}{2\sigma^2}\|w\|^2 \end{aligned}$$

3. We use a (very robust) student t likelihood with a mean of $w^T x_i$ and ν degrees of freedom, and a Gaussian prior with a mean of μ_j and a variance of λ^{-1},

$$p(y_i \mid x_i, w) = \frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu\pi}\Gamma\left(\frac{\nu}{2}\right)}\left(1 + \frac{(w^T x_i - y_i)^2}{\nu}\right)^{-\frac{\nu+1}{2}}, \quad p(w_j) = \sqrt{\frac{\lambda}{2\pi}}\exp\left(-\frac{\lambda}{2}(w_j - \mu_j)^2\right).$$

where Γ is the gamma function (which is always non-negative). You can use μ as a vector whose components are μ_j.

Answer:

$$\begin{aligned}
f(w) &= -\log(p(X, w \mid y)) = -\log(p(y \mid X, w) \cdot p(w)) \\
&= -\log\left(\prod_i p(y_i \mid x_i, w)\cdot\right) - \log\left(\prod_j p(w_j)\right) \\
&= -\sum_i \left(\log\left(\frac{\Gamma\left(\frac{\nu+1}{2}\right)}{\sqrt{\nu}\Gamma\left(\frac{\nu}{2}\right)}\right) - \frac{\nu+1}{2}\log\left(1 + \frac{(w^T x_i - y_i)^2}{\nu}\right)\right) - \sum_j \left(\log\left(\sqrt{\frac{\lambda}{2\pi}}\right) - \frac{\lambda}{2}(w_j - \mu_j)^2\right) \\
&\sim \frac{\nu+1}{2}\sum_i \left(\log\left(1 + \frac{(w^T x_i - y_i)^2}{\nu}\right)\right) + \frac{\lambda}{2}\|w - \mu\|^2
\end{aligned}$$

4. We use a Poisson-distributed likelihood (for the case where y_i represents counts), and a uniform prior for some constant κ,

$$p(y_i | w^T x_i) = \frac{\exp(y_i w^T x_i)\exp(-\exp(w^T x_i))}{y_i!}, \quad p(w_j) \propto \kappa.$$

(This prior is "improper", since $w \in \mathbb{R}^d$ but κ doesn't integrate to 1 over this domain. Nevertheless, the posterior will be a proper distribution.)

Answer:

$$\begin{aligned}
f(w) &= -\log(p(X, w \mid y)) = -\log(p(y \mid X, w) \cdot p(w)) \\
&= -\log\left(\prod_i p(y_i \mid x_i, w)\cdot\right) - \log\left(\prod_j p(w_j)\right) \\
&= -\sum_i \left(y_i w^T x_i - \exp(w^T x_i) - \log(y_i!)\right) - \sum_j \left(\log(\kappa)\right) \\
&\sim -y^T X w + \|\exp(Xw)\|_1
\end{aligned}$$

3 Principal Component Analysis [19 points]

3.1 PCA by Hand [6 points]

Consider the following dataset, containing 5 examples with 3 features each:

x_1	x_2	x_3
0	2	0
3	-4	3
1	0	1
-1	4	-1
2	-2	2

Recall that with PCA we usually assume we centre the data before applying PCA (so it has mean zero). We're also going to use the usual form of PCA where the PCs are normalized ($\|w\| = 1$), and the direction of the first PC is the one that minimizes the orthogonal distance to all data points.

1. What is the first principal component?

   Answer: After centering the data:

   $$X = \begin{bmatrix} -1 & 2 & -1 \\ 2 & -4 & 2 \\ 0 & 0 & 0 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{bmatrix}$$

   We notice that all the columns are multiples of each other, meaning that all 5 examples lie along a straight line in 3D feature space. Therefore, the first un-normalized PC is:

   $$w_1 \propto \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix}$$

   Which after normalizing is:

   $$w_1 = \frac{1}{\sqrt{6}} \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix}$$

2. What is the reconstruction loss (L2 norm squared) of the point $(2.5, -3, 2.5)$? (Show your work.)

   Answer: The reconstructed point is:

   $$(w_1 \cdot (2.5, -3, 2.5))\, w_1 = \frac{11}{6}(1, -2, 1)$$

   And so the loss is:

   $$2\left(2.5 - \frac{11}{6}\right)^2 + \left(-3 + \frac{22}{6}\right)^2 = \frac{4}{3}$$

3. What is the reconstruction loss (L2 norm squared) of the point $(1, -3, 2)$? (Show your work.)

   Answer: The reconstructed point is:

   $$(w_1 \cdot (1, -3, 2))\, w_1 = \frac{4}{3}(1, -2, 1)$$

   And so the loss is:

   $$\left(1 - \frac{4}{3}\right)^2 + \left(-3 + \frac{8}{3}\right)^2 + \left(2 - \frac{4}{3}\right)^2 = \frac{2}{3}$$

Hint: it may help (a lot) to plot the data before you start this question.

3.2 Data Visualization [7 points]

If you run `python main.py -q 3.2`, the program will load a dataset containing 50 examples, each representing an animal. The 85 features are traits of these animals. The script standardizes these features and gives two unsatisfying visualizations of it. First, it shows a plot of the matrix entries, which has too much information and thus gives little insight into the relationships between the animals. Next it shows a scatterplot based on two random features and displays the name of 15 randomly-chosen animals. Because of the binary features even a scatterplot matrix shows us almost nothing about the data.

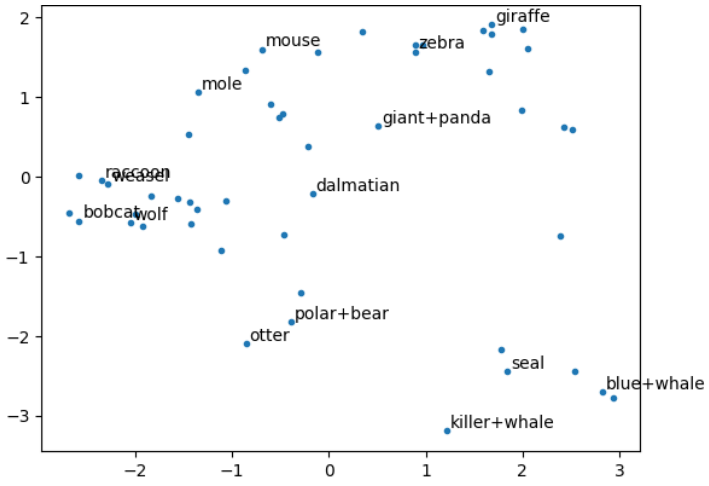
In `compressors.py`, you will find a class named `PCA`, which implements the classic PCA method (orthogonal bases via SVD) for a given k, the number of principal components. Using this class, create a scatterplot that uses the latent features z_i from the PCA model with $k = 2$. Make a scatterplot of all examples using the first column of Z as the x-axis and the second column of Z as the y-axis, and use `plt.annotate()` to label the points corresponding to `random_is` in the scatterplot. (It's okay if some of the text overlaps each other; a fancier visualization would try to avoid this, of course, but hopefully you can still see most of the animals.) Do the following:

1. Hand in your modified demo and the scatterplot.

   Answer: https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.annotate.html

```
encoder = PCAEncoder(2)
encoder.fit(X_train)
Z = encoder.encode(X_train)

fig, ax = plt.subplots()
ax.plot(Z[:, 0], Z[:, 1], ".")
for i in random_is:
    ax.annotate(animal_names[i],
                xy=Z[i], xycoords="data",
                xytext=(2, 2), textcoords="offset points")
utils.savefig("animals_pca.png", fig)
```



2. Which trait of the animals has the largest influence (absolute value) on the first principal component?

   Answer: paws

3. Which trait of the animals has the largest influence (absolute value) on the second principal component?

   Answer: vegetation

3.3 Data Compression [6 points]

It is important to know how much of the information in our dataset is captured by the low-dimensional PCA representation. In class we discussed the "analysis" view that PCA maximizes the variance that is explained by the PCs, and the connection between the Frobenius norm and the variance of a centred data matrix X. Use this connection to answer the following:

1. How much of the variance is explained by our two-dimensional representation from the previous question?

   Answer: $\approx 18\%$

2. How many PCs are required to explain 50% of the variance in the data?

   Answer: At least 12 PCs.

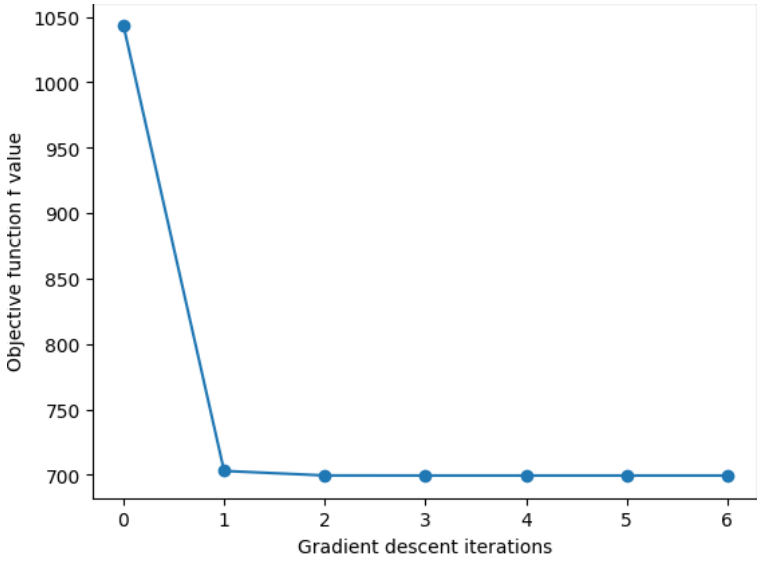
Note: you can compute the Frobenius norm of a matrix using the function `np.linalg.norm`, among other ways. Also, note that the "variance explained" formula from class assumes that X is already centred.

4 Stochastic Gradient Descent [20 points]

If you run `python main.py -q 4`, the program will do the following:

1. Load the dynamics learning dataset ($n = 10000, d = 5$)
2. Standardize the features
3. Perform gradient descent with line search to optimize an ordinary least squares linear regression model
4. Report the training error using `np.mean()`
5. Produce a learning curve obtained from training

The learning curve obtained from our `GradientDescentLineSearch` looks like this:



This dataset was generated from a 2D bouncing ball simulation, where the ball is initialized with some random position and random velocity. The ball is released in a box and collides with the sides of the box, while being pulled down by the Earth's gravity. The features of X are the position and the velocity of the ball at some timestep and some irrelevant noise. The label y is the y-position of the ball at the next timestep. Your task is to train an ordinary least squares model on this data using stochastic gradient descent instead of the deterministic gradient descent.

4.1 Batch Size of SGD [5 points]

In `optimizers.py`, you will find `StochasticGradient`, a *wrapper* class that encapsulates another optimizer– let's call this a base optimizer. `StochasticGradient` uses the base optimizer's `step()` method for each mini-batch to navigate the parameter space. The constructor for `StochasticGradient` has two arguments: `batch_size` and `learning_rate_getter`. The argument `learning_rate_getter` is an object of class `LearningRateGetter` which returns the "current" value learning rate based on the number of batch-wise gradient descent iterations. Currently. `ConstantLR` is the only class fully implemented.

Submit your code from `main.py` that instantiates a linear model optimized with `StochasticGradient` taking `GradientDescent` (not line search!) as a base optimizer. Do the following:

1. Use ordinary least squares objective function (no regularization).
2. Using `ConstantLR`, set the step size to $\alpha^t = 0.0003$.

3. Try the batch size values of `batch_size` $\in \{1, 10, 100\}$.

For each batch size value, use the provided training and validation sets to compute and report training and validation errors after 10 epochs of training. Compare these errors to the error obtained previously.

Answer: The training/validation errors for batch sizes 1, 10 are identical to the errors from `GradientDescentLineSearch`. But at the higher batch size of 100 (corresponding to a lower number of optimization steps), `StochasticGradient` starts to perform worse.

- `GradientDescentLineSearch`

  Training MSE: 0.140

  Validation MSE: 0.140

- `StochasticGradient` - batch size 1

  Training MSE: 0.140

  Validation MSE: 0.140

- `StochasticGradient` - batch size 10

  Training MSE: 0.140

  Validation MSE: 0.140

- `StochasticGradient` - batch size 100

  Training MSE: 0.178

  Validation MSE: 0.177

```
for bs in [1, 10, 100]:
    loss_fn = LeastSquaresLoss()
    optimizer = StochasticGradient(GradientDescent(), ConstantLR(0.0003), bs, max_evals=10)
    model = LinearModel(loss_fn, optimizer)
    model.fit(X_train, y_train)

    print(f"\nBatch size: {bs}")
    print(f"Training MSE: {((model.predict(X_train) - y_train) ** 2).mean():.3f}")
    print(f"Validation MSE: {((model.predict(X_val) - y_val) ** 2).mean():.3f}")
```

4.2 Learning Rates of SGD [6 points]

Implement the other unfinished `LearningRateGetter` classes, which should return the learning rate α^t based on the following specifications:

1. `ConstantLR`: $\alpha^t = c$.
2. `InverseLR`: $\alpha^t = c/t$.
3. `InverseSquaredLR`: $\alpha^t = c/t^2$.
4. `InverseSqrtLR`: $\alpha^t = c/\sqrt{t}$.

Submit your code for these three classes.

```
class InverseLR(LearningRateGetter):
    def get_learning_rate(self):
        self.num_evals += 1
        return self.multiplier / self.num_evals
```

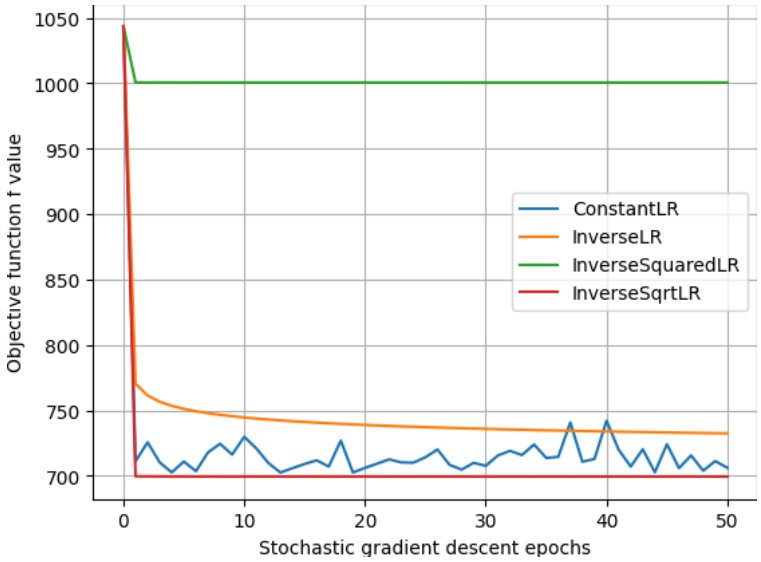
```
class InverseSquaredLR(LearningRateGetter):
    def get_learning_rate(self):
        self.num_evals += 1
        return self.multiplier / self.num_evals ** 2

class InverseSqrtLR(LearningRateGetter):
    def get_learning_rate(self):
        self.num_evals += 1
        return self.multiplier / np.sqrt(self.num_evals)
```

4.3 The Learning Curves (Again) [9 points]

Using the four learning rates, produce a plot of learning curves visualizing the behaviour of the objective function f value on the y-axis, and the number of stochastic gradient descent epochs (at least 50) on the x-axis. Use a batch size of 10. Use $c = 0.1$ for every learning rate function. Submit this plot and answer the following question. Which step size functions lead to the parameters converging towards a global minimum?

Answer: Both `InverseLR` and `InverseSqrtLR` step size functions lead to convergence to a global minimum, but using `InverseSqrtLR` we converge much faster.



5 Very-Short Answer Questions [18 points]

Answer each of the following questions in a sentence or two.

1. Assuming we want to use the original features (no change of basis) in a linear model, what is an advantage of the "other" normal equations over the original normal equations?

   Answer: They are faster when $n < d$. The cost of solving the "other" normal equations is $O(n^2d+n^3)$ instead of the cost $O(nd^2 + d^3)$ of solving the original.

2. In class we argued that it's possible to make a kernel version of k-means clustering. What would an advantage of kernels be in this context?

   Answer: They allow us to have non-convex clusters by defining a distance basis different from the regular Euclidean distance.

3. In the language of loss functions and regularization, what is the difference between MLE and MAP?

   Answer: MLE given a likelihood is equivalent to minimizing a loss function. MAP given a likelihood and a prior is equivalent to minimizing a regularized loss function.

4. What is the difference between a generative model and a discriminative model?

   Answer: A generative model models both X (input) and y (output), and therefore would optimize $p(X, y \mid w)$. A discriminative model assumes X is fixed and only optimizes $p(y \mid w, X)$.

5. In this course, we usually add an offset term to a linear model by transforming to a Z with an added constant feature. How can we do that in a kernel model?

   Answer: Simply by adding 1 to the kernel.

   $$k(x_i, x_j) = 1 + x_i^T x_j$$

6. With PCA, is it possible for the loss to increase if k is increased? Briefly justify your answer.

   Answer: No. Adding a PC (i.e. increasing k) at-worst adds an extra degree of freedom to our model which always decreases the loss.

7. Why doesn't it make sense to do PCA with $k > d$?

   Answer: Our examples live in a d-dimensional feature space. With $k = d$, the span of the orthogonal PCs already covers the entire feature space. An additional PC is necessarily linearly dependent on the other PCs.

8. In terms of the matrices associated with PCA (X, W, Z, $\hat{X}$), where would a single "eigenface" be stored?

   Answer: A single "eigenface" is one row of the W matrix.

9. What is an advantage and a disadvantage of using stochastic gradient over SVD when doing PCA?

   Answer: The advantage is it usually has a lower computational cost. The disadvantage is finding the optimal solution is not guaranteed.