Lab 2 - Machine Learning Basics

ARIZONA STATE UNIVERSITY
SCHOOL OF ELECTRICAL, COMPUTER,
AND ENERGY ENGINEERING,
EEE598: Deep Learning Media Processing & Understanding

1 Objectives

In this lab we will implement several basic machine learning models. Specifically we will:

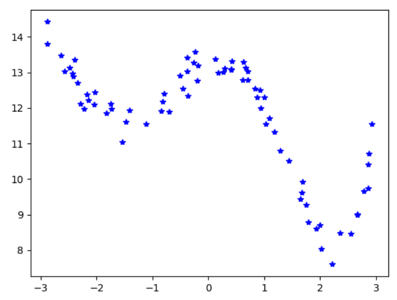
- Implement Least Squares Fitting in Python
- Implement K-Nearest neighbors (KNN) in Python
- Implement Logistic Regression in Python
- Implement Support Vector Machines (SVM) in Python
- Learn how to use Scikit-Learn models

2 Brief Machine Learning Introduction

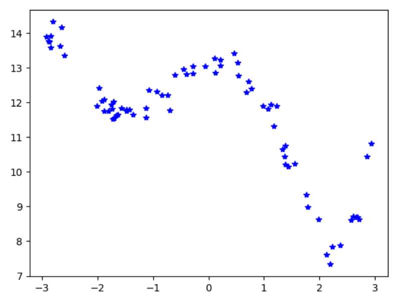
A machine learning model takes the general form of $f(\mathbf{x};\theta) = \mathbf{y}$, where $\mathbf{x}$ contains the input features, $\mathbf{y}$ contains the corresponding class labels, and θ represents the parameters of the model. f is the machine learning model which could take many forms, from the simple linear classifiers we will implement in this lab, to layered deep networks we will implement in later labs. The choice of the model is largely problem dependent. $\mathbf{x}$ contains several data points $\mathbf{x_i}$ where each data point is a vector of "features". The features could be some measurements of the data, such as image pixels, physical measurements, word embeddings etc. The choice of features is problem dependent, and for some problems the choice of features is more important than the choice of machine learning model. The label $\mathbf{y}$ could be a category label, a continuous valued number, or even a vector of numbers. In this lab $\mathbf{y}$ is constrained to be either -1 or 1 (a binary classification problem). The model parameters θ must be optimized based on the given training data $(\mathbf{x}_{train}, \mathbf{y}_{train})$. After this optimization procedure, we say that the model has been "trained". We can then test the model on new data $\mathbf{x}_{test}$.

For machine learning we must be careful to separate training data and testing data. We give training data to the model to learn parameters, but we never give testing data to the model for this purpose. If testing data is accidentally used for training, model performance may be misleadingly high on the testing data. It is possible for some machine learning models to obtain perfect performance on the training set (K-Nearest neighbors can do this). So if testing data is used for training, the performance on the testing dataset may be perfect, but if new data is collected the performance will not be perfect. The goal of the separate testing set is to get an idea of how the model would work "in the real world" with new, unseen data.

We use training data to optimize the parameters of the network, but there is also the choice of the model f that we must "optimize". Even for a single model type there is often one or



(a) Train data



(b) Test data

Figure 1: Data for polynomial fitting

more parameters of the model. These parameters often cannot be easily optimized with numerical optimization methods. We optimize them “by hand” by selecting several values and seeing what works best, or by using a grid search. We term these parameters “hyper-parameters” to distinguish them from the learnable parameters in the model. However if we use the testing set to optimize the hyper-parameters, the model may have unfairly achieved higher performance on the testing set. It is for this reason that we usually use a completely separate testing set to optimize hyper-parameters. We call this the *validation set*. After choosing good parameters on the validation set, we finally test on the testing set to get a measure of real world performance. No parameters or hyper-parameters have been tuned to the testing set, so it is an unbiased estimate of performance. In this lab we will not be doing much hyper-parameter optimization so for simplicity we will only consider a training and testing set. For real world problems, it is recommended to use a validation set as well.

3 Least Squares and Model Capacity

To start, we will implement least squares fitting of a polynomial model. In this example we consider a regression problem where we want to predict real values y for a given x. To do this we will use a polynomial model of the form $y = c_0 + c_1x + c_2x^2 + c_3x^3 + \dots c_kx^k$. We want to learn the values of c_i to achieve the best fit. The key hyper-parameter of this model is the degree of the polynomial k. With a larger degree, we can model more complicated functions, but we might overfit the data. Conversely, a polynomial degree that is too small will not be able to fit the data well.

Figure 1 shows the given data. We have separate training data and testing data that are generated using the same process, but because of noise the training and testing data are not the same. We would like our polynomial function to fit the data well, but without fitting the noise. We will use the training data to fit the coefficients of the polynomial model, and then test the performance of the model on the test data. The data is provided in a file called `ls_data.pkl`. This file stores a dictionary of Numpy vectors using the Python `pickle` package. This data can be loaded with the following code:

```
data = pickle.load(open("ls_data.pkl", "rb"))
x_train = data['x_train']
x_test = data['x_test']
y_train = data['y_train']
y_test = data['y_test']
```

We will implement the least squares fitting model as a Python class called `LeastSquares` in the file `ls.py`. We also provided tests in the `test_ls.py` file. Use these tests to make sure your implementation is correct. The class has three functions `__init__`, `fit`, and `predict`. The `__init__` function is provided. This function simply initializes the class with some provided polynomial degree `k`:

```
def __init__(self, k):
    """
    Initialize the LeastSquares class

    The input parameter k specifies the degree of the polynomial
    """
    self.k = k
    self.coeff = None
```

The `fit` function takes some data and fits the parameters of our model using least squares. In least squares we consider the problem $\mathbf{Ax} = \mathbf{b}$, and solve for $\mathbf{x}$. To avoid confusion with the previously defined x, we will rewrite this as $\mathbf{Ac} = \mathbf{y}$, where $\mathbf{c}$ is the vector of polynomial coefficients for which we want to find, and $\mathbf{y}$ are the target outputs we will use for fitting.

The solution to this least squares problem is $\mathbf{c} = \mathbf{A}^{\dagger}\mathbf{y}$, where † is the pseudo-inverse. To adapt this to our polynomial fitting problem we need to define the matrix $\mathbf{A}$ appropriately. Since $\mathbf{c}$ is the polynomial coefficients, we can see that $\mathbf{A}$ should contain the polynomial forms of our data. $\mathbf{A}$ is defined as:

$$\mathbf{A} = \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^k \\ 1 & x_1 & x_1^2 & \dots & x_0^k \\ \dots & \dots & \dots & \dots & \dots \\ 1 & x_{N-1} & x_{N-1}^2 & \dots & x_{N-1}^k \end{bmatrix} \tag{1}$$

where x_i is the ith sample of the input in $\mathbf{x}$.

We can use the Numpy function `np.linalg.pinv` to compute the pseudo-inverse. It is important that we use the pseudo-inverse instead of the inverse because we have an over-determined system, and A is not invertible.

In the `predict` function we can take the computed $\mathbf{A}$ and stored $\mathbf{c}$ (in the `self.coeff` variable) and compute the outputs using the function $\mathbf{y} = \mathbf{Ac}$. The predict function should return this output.

After implementing the class in the `ls.py` file, we can run the model by creating a new Python script `run_ls.py` to load the data, train, and test the model. Keeping the model implementation separate from the `run_ls.py` code allows us to easily reuse our model for other applications. In `run_ls.py` we call the `fit` function using the training data to fit the parameters of the model, and then call the `predict` function using the testing data. To assess the performance of the model we can measure the mean square error (MSE) of the testing data. Recall that mean square error is defined as:

$$\text{MSE}(\mathbf{y}, \mathbf{z}) = \frac{1}{N} \sum_{i=0}^{N-1} (y_i - z_i)^2 \tag{2}$$

We have provided a starting point for `run_ls.py`. This code loads the data, trains the model, and gets the test predictions. You need to modify this code to compute the MSE of the test and train data. Also we want to test the performance of the model as we vary the degree of the polynomial from 1 to 20. Plot the train and test MSE as a function of the polynomial degree using Matplotlib. By plotting the train and test error, we can see for which polynomial degrees the model is underfitting (train and test error are both high), or overfitting (train error is low, but test error is high).

Deliverable: Least Square Polynomial Fitting

- A working version of `ls.py` that passes all of the automated unit tests in `test_ls.py`
- A plot of the train error and test error vs the degree of the polynomial saved as `ls_error.png`
- A modified working version of `run_ls.py` that plots the train error and test error vs the degree of the polynomial.

4 Implement K-Nearest Neighbors Model (KNN)

Next we will implement is the K-Nearest Neighbors model (KNN). For this model, training involves memorizing the entire training set. During testing, the KNN model simply compares the test sample to its k nearest neighbors in the training set using some distance measure in the feature space, where k is a hyper-parameter of the model. The predicted output is the mode of the nearest neighbors' classes. For example, if $k = 1$ the model finds the nearest training sample in the feature space, and predicts the same class as that training sample. Figure 2 shows an example of KNN for $k = 3$. For our implementation of KNN, we will use the Euclidean distance measure in the feature space, however it is possible to use any distance measure in KNN.

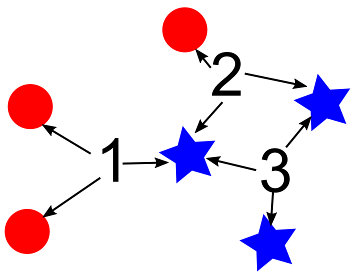


Figure 2: KNN toy example. The blue stars correspond to the training data from one class, and the red circles correspond to the training data from another class. The numbers are new test data points. For each test data point, we predict the label using the k nearest neighbors from the training data. For $k = 3$, 1 is mapped to the red circle class, and 2 and 3 are mapped to the blue star class. But note that if $k = 1$, 1 and 3 are mapped to the blue star class, and 2 is mapped to the red circle class.

Implementation We will implement a simple version of the KNN model. Our implementation will take the form of a Python class. By using a class, we can train the model once, and then use the trained object to test many times. Of course "training" in KNN is simply storing the training data in some class variables. We will implement the class using an interface similar to the `Scikit-Learn` models we will use later. By using a common class interface (i.e. functions have the same names and same arguments), we can swap model objects in our code to compare the performance of different models with minimal effort.

We provided the initialization routine of the class in the file `knn.py`. Note the presence of the optional keyword argument in the function definition. This means that if the class is initialized using the code `KNN()`, then the `k` variable will be set to the default value 3. But we can also call the code and explicitly set `k` using `KNN(k=5)`. We set the `x_train` and `y_train` variables to `None`. It is good practice to initialize these variables in the initialization function, even if the variables are initialized to `None`. This allows us to later check if the model has been trained by checking if `self.x_train` `is not None`. Also defining all the class variables in the initialization function is good for code readability.

```
    def __init__(self, k=3):
        self.x_train = None
        self.y_train = None
        self.k = k
```

You will need to implement the remaining `predict(x)` and `fit(x,y)` functions in the `knn.py` file. The `fit(x,y)` function takes two arguments: the training data `x` and the training labels `y`. We will be using the Scikit-learn convention for the input data format. The input variable `x` is a $N \times D$ matrix where N is the number of data samples, and D is the number of feature dimensions. In other words, the rows of `x` correspond to data samples and the columns correspond to features. The variable `y` is a vector (or equivalently a Numpy matrix with one dimension) of size N. A typical fit function would run some optimization procedure to find parameters of the model. For the KNN case, the parameters of the model are the training data itself, so we only need to store the training data in the `fit(x,y)` function.

Most of the implementation occurs in the `predict(x)` function. For a given testing data point, we need to compute the Euclidean distance between the testing point and all of the training data points. The k closest points using these distances are the k-nearest neighbors. We can use the `np.argsort` function to get the indices of the neighbors sorted by distance. Using the nearest k indices, we can find the class labels of the k-nearest neighbors by using the stored labels in the `self.y_train` variable. Finally, of the k-nearest class labels we take the most common label as the class prediction. This can be computed using `scipy.stats.mode`. The input `x` in the `predict(x)` function is of the same format as in the training function ($N \times D$). This means that the `predict(x)` function should handle inputs with multiple data samples, and should return a vector of size N of the predicted class labels.

Extra Information: KNN Speed

The naïve implementation of the KNN model must loop through every data point in the training set to compute the distances for a single test point. The computational complexity of testing a single sample grows with the size of the dataset. This is clearly not desirable as modern datasets may have thousands or millions of samples. However, we do not need to check every sample in the training dataset every time we test a sample in KNN. There are algorithms which attempt to partition the search space in order to make prediction more efficient. The KD-tree is one such popular algorithm that can be used to speed up KNN [1].

We have provided unit tests in `test_knn.py`. This will use some synthetic data to make sure the nearest neighbors computation is correct. For running on a dataset we do not use unit testing, because it is difficult to know what the output should be for a large dataset before training the model. For this we have provided a small script `run_knn.py` that shows how to train and test on a dataset. We will use a synthetic dataset (Figure 3) to test our model because it is easy to visualize the input features. The synthetic dataset is implemented in the `datasets.py` which we can import to test our code. The dataset consists of two classes that are randomly generated from two 2-d Gaussian distributions. The classes slightly overlap which makes the learning problem difficult because some data points are ambiguous. In a real world problem, the two dimensions could correspond to two different features of the data. For example, the x axis could be height of an individual, the y axis could represent the peak running speed of that individual. The label could be whether or not that person played basketball in high school. Although the input features should be good indicators of the output label, there may not be a clear separation with these features. For example, there could be some people that are tall and fast, yet didn't play basketball.

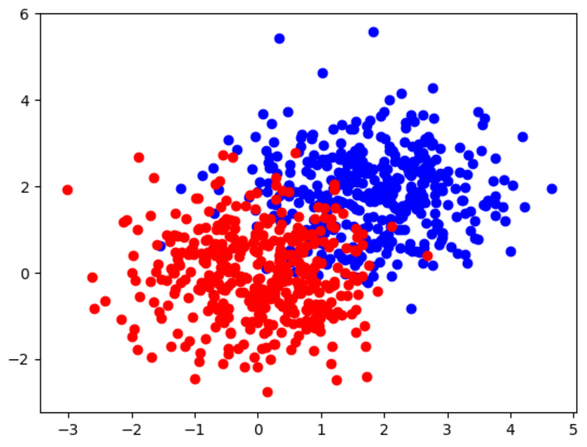


Figure 3: Toy classification dataset. Blue dots represent class $y = -1$ and red dots represent class $y = +1$. The classes are not perfectly separable which makes classification difficult. Real world problems often have similar noise which makes perfect classification difficult.

run_knn.py

```
from knn import KNN
import numpy as np
import datasets

# load data
x_train, y_train, x_test, y_test = datasets.gaussian_dataset(n_train
    =800, n_test=800)

model = KNN(k=3)
model.fit(x_train, y_train)

y_pred = model.predict(x_test)
print("knn accuracy: " + str(np.mean(y_pred == y_test)))
```

Deliverable: KNN Implementation

- A working version of `knn.py` that passes all of the automated unit tests in `test_knn.py`
- A plot of the classification test accuracy on the Gaussian data vs the k parameter (vary from 1 to 51 in steps of 5 saved as `knn_k.png`
- A modified working version of `run_knn.py` that plots the test accuracy on the Gaussian data vs the k parameter.

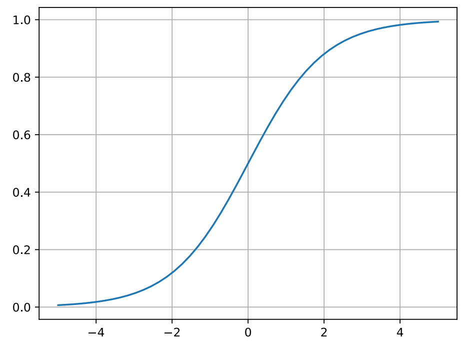


Figure 4: Logistic function $f(x) = \frac{1}{1+exp(-x)}$. The logistic function forces the output to be between 0 and 1. This allows us to use the output as a probability.

5 Implement Logistic Regression

Background Next we will implement *logistic regression* [2]. Despite the name, logistic regression is actually used as a classification model. Logistic regression is a binary classification model, meaning that it can handle only two classes ($y = -1$ and $y = +1$). Although this may seem restrictive, it is easy to extend a binary classification model to a multi-class model [3].

Different from KNN, logistic regression outputs probabilities instead of hard class labels. This is why the name contains the word "regression" instead of "classification". Lets define the output of the logistic regression model as $f(x)$, where $0 <= f(x) <= 1$. If $f(x)$ is close to 1, then the model predicts a high probability of class 1. Likewise, if $1-f(x)$ is close to 1 then the model predicts a high probability of class 0. The probabilities give us more intuitive explanation of the prediction for a test example. For a KNN prediction, we don't know whether the data sample is very representative of a particular class or is somewhere between the two classes. With logistic regression, if $f(x)$ is near 0.5 then the data sample has characteristics of both classes which makes it difficult to classify. In certain applications this probability is very useful, because we might want to include human judgment for these difficult to classify samples. A probability output also allows us to use logistic regression in conjunction with Bayes' rule, which allows us to design priors for our problem. Similar to logistic regression, deep neural networks for classification problems also output probabilities instead of hard labels.

Logistic regression is a simple linear classifier, that learns a function with a vector weight parameter $\mathbf{w}$ and a scalar bias term b. In terms of the input data this linear function is $\mathbf{w}^T\mathbf{x} + b$. To achieve the probabilistic output, logistic regression uses a logistic function to "squash" the linear output (Figure 4). This function ensures that the output is always between 0 and 1. The complete model for the logistic regression takes the form:

$$f(\mathbf{x_i}) = \frac{1}{1 + exp(-(\mathbf{w}^T\mathbf{x_i} + b))} \tag{3}$$

Now lets assume that we have some data $\mathbf{x}$ and some corresponding labels $\mathbf{y}$. We assume that we have a binary classification problem where y_i can be either -1 or $+1$. To optimize the parameters we need to define a *loss function* to minimize. The loss function is sometimes referred to as a *cost function*. We use the cross entropy loss function which for two class is defined as:

$$\ell(\mathbf{x}, \mathbf{y}) = \frac{1}{N}\sum_i -\mathbb{I}(y_i = 1)ln(f(\mathbf{x}_i)) - \mathbb{I}(y_i = -1)ln(1 - f(\mathbf{x}_i)) \tag{4}$$

where N is the number of data samples, i is an index to a particular sample, and $\mathbb{I}$ is an indicator function. The indicator function is 1 if the argument is true, and 0 otherwise. Basically this loss function says that when the label $y_i = 1$, we want the corresponding value of $f(x_i)$ to be high. Similarly, when $y_i = -1$ we want $f(x_i)$ to be low. After some algebra, this loss can be simplified for our problem to:

$$\ell(\mathbf{x}, \mathbf{y}) = \frac{1}{N} \sum_i ln(1 + exp(-y_i(\mathbf{w}^T \mathbf{x}_i + b))) \tag{5}$$

Gradients There are many ways to minimize the loss function we just defined (e.g. see [4]). We will use the simplest approach and use gradient descent to minimize the loss. In gradient descent we compute the gradient of the loss with respect to each learnable parameter of our model. We then perturb each parameter by small step in the opposite direction of the gradient. For this problem we have two parameters that we adjust according to:

$$b' = b - \eta \left(\frac{d\ell}{db} \right) \tag{6}$$

$$\mathbf{w}' = \mathbf{w} - \eta \left(\frac{d\ell}{d\mathbf{w}} \right) \tag{7}$$

where η is the *learning rate*. Note that we are subtracting the derivative because we want to minimize the loss function (i.e. we want to "go downhill"). The learning rate governs how big of a step is taken in the descent direction. We might be tempted to set a large learning rate to try to make the optimization converge faster. However if the rate is too large, the gradient descent update might overshoot the function that is being minimized which might cause slow convergence or no convergence at all. A small learning rate won't have this overshooting problem. However, if the learning rate is too small the optimization may get trapped in a small local minima, because the step is not large enough to "climb out" of the local minima. In practice the learning rate is a hyper-parameter that must be appropriately set for the problem.

To do the gradient descent update we need to compute the gradients of the loss function with respect to our learnable parameters. To do this we can use the chain rule on equation 5:

$$\begin{aligned} f &= -y_i(\mathbf{w}^T \mathbf{x}_i + b) \\ g &= 1 + exp(f) \\ h &= \frac{1}{N} \sum_i ln(g) \\ \frac{d\ell(\mathbf{x})}{db} &= \left(\frac{dh}{dg}\right)\left(\frac{dg}{df}\right)\left(\frac{df}{db}\right) \\ \frac{d\ell(\mathbf{x})}{d\mathbf{w}} &= \left(\frac{dh}{dg}\right)\left(\frac{dg}{df}\right)\left(\frac{df}{d\mathbf{w}}\right) \end{aligned} \tag{8}$$

Computing these derivatives is relatively simple:

$$\begin{aligned} \frac{df}{d\mathbf{w}} &= -y_i \mathbf{x}_i \\ \frac{df}{db} &= -y_i \\ \frac{dg}{df} &= exp(-y_i(\mathbf{w}^T \mathbf{x}_i + b)) \\ \frac{dh}{dg} &= \frac{1}{N} \sum_i \frac{1}{1 + exp(-y_i(\mathbf{w}^T \mathbf{x}_i + b))} \end{aligned} \tag{9}$$

Finally we get the two derivatives we are interested in:

$$\frac{d\ell(\mathbf{x},\mathbf{y})}{db} = \frac{1}{N}\sum_i \frac{-y_i}{1+exp(+y_i(\mathbf{w}^T\mathbf{x}_i+b))} \tag{10}$$

$$\frac{d\ell(\mathbf{x},\mathbf{y})}{d\mathbf{w}} = \frac{1}{N}\sum_i \frac{-y_i\mathbf{x}_i}{1+exp(+y_i(\mathbf{w}^T\mathbf{x}_i+b))} \tag{11}$$

The gradient descent method is a repeated application of Eq. 7 (Algorithm 1). For the logistic regression implementation, we will compute the average gradient using the entire training dataset. One pass through the entire dataset is called an *epoch*. Updating using the entire dataset is feasible because our training dataset is relatively small. However if our training set is larger, we can split the training set into smaller batches and perform a gradient descent update after each batch. We can check the loss after each epoch to ensure that it is decreasing. Figure 5 shows the logistic regression model decision boundary on the toy Gaussian dataset. As we train for more epochs the decision boundary makes a better separation between the two classes.

Algorithm 1: Gradient Descent Algorithm

```
  Input  : Training data $(\mathbf{x},\mathbf{y})$
1 $\mathbf{w} \leftarrow$ randomly initialized
2 $b \leftarrow 0$
3 for $i$ in range(n_epochs) do
4     $b \leftarrow b - \eta\frac{d\ell(\mathbf{x},\mathbf{y})}{db}$
5     $\mathbf{w} \leftarrow \mathbf{w} - \eta\frac{d\ell(\mathbf{x},\mathbf{y})}{d\mathbf{w}}$
6 end
```

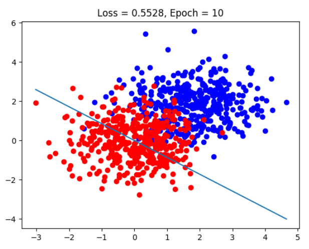
After training we can predict the output for new test data. Equation 3 gives us a continuous number representing the probability of class $y = 1$. In this case, we want to predict discrete class labels. This can be done by thresholding the output of Eq. 3:

$$y_{test} = \begin{cases} -1 & f(\mathbf{x}_{test}) \leq 0.5 \\ +1 & f(\mathbf{x}_{test}) > 0.5 \end{cases} \tag{12}$$

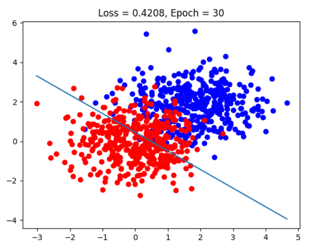
Implementation We will be implementing the logistic regression model in the script we offered `logistic_regression.py` using the gradient expressions we just derived. The structure of the class is similar to our KNN implementation, however we have additional methods for computing the gradients and the loss. As usual, the unit tests for the logistic regression model are in the `test_logistic_regression.py` file. Now the unit testing is starting to become useful, because we need to implement several functions for the model to train properly, and if any of these functions are incorrect, our model may not train. The unit tests allow us to pinpoint possible reasons for why the model is not training.

Table 1 provides descriptions of the methods of the `LogisticRegression` class. Table 2 shows the expected dimensions of the variables in this class. The variables in your implementation should have the same dimensions. Recall that in math, vectors are by convention column vectors, however in machine learning implementations it is convention to express data in terms of row vectors. Like in the KNN implementation, the input variable `x` has data samples as rows and features as columns. A single sample `x[i,:]` is a row vector. To deal with this difference between the math and the implementation you can make use of the `np.transpose` function. To check the dimensions of a Numpy variable you can use `ndarray.shape`.

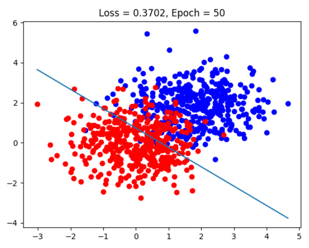
We will be using the same toy dataset that we used for the KNN implementation. We can copy the `run_knn.py` to create a `run_logreg.py` file. Since we implement the `LogisticRegression` class



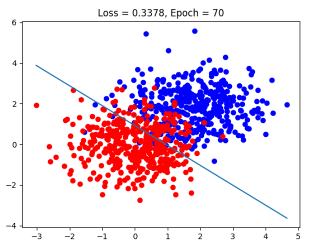
(a) Iteration 10



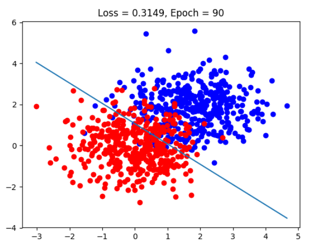
(b) Iteration 30



(c) Iteration 50



(d) Iteration 70



(e) Iteration 90

Figure 5: Convergence of logistic regression. Here we show the learned decision boundary at different stages of training with a learning rate of 0.1. Each epoch is one gradient descent update computed over the entire training data. We plot the decision boundary that separates the predicted positive class from the predicted negative class. As gradient descent progresses, the decision boundary comes closer to the optimal boundary.

Table 1: Functions in `logistic_regression.py`

Function	Description
`__init__`	Initialization of class (provided)
`forward`	Implement logistic function (Eq. 3)
`loss`	Implement logistic loss function (Eq. 5)
`grad_loss_wrt_b`	Compute gradient of loss with respect to b (Eq. 10)
`grad_loss_wrt_w`	Compute gradient of loss with respect to $\mathbf{w}$ (Eq. 11)
`fit`	Run gradient descent on given data to optimize parameters (Algorithm 1)
`predict`	Predict output given data (Eq. 12)

with an identical interface as the `KNN` model, all we need to do is import the `LogisticRegression` class and change the line that defines the model variable. The `fit` and `predict` functions are called in exactly the same way as in the KNN example.

Adding Regularization One potential problem with the logistic regression model we implemented is that there is no way to limit the magnitude of $\mathbf{w}$. During optimization the magnitude of $\mathbf{w}$ can become arbitrarily large. This may cause "over-fitting" to the training data by over emphasizing a dimension of the input by learning a very large weight. It is common practice to add a regularization term to the cost function that limits the magnitude of $\mathbf{w}$:

$$\ell(\mathbf{x}, \mathbf{y}) = \frac{1}{N} \sum_i ln(1 + exp(-y_i(\mathbf{w}^T \mathbf{x}_i + b))) + \frac{1}{2}\lambda \mathbf{w}^T \mathbf{w} \tag{13}$$

Table 2: Variables in `logistic_regression.py`

Variable	Description	Dimensions (rows × cols)
`x`	Input data	$(N \times D)$
`y`	Input labels	(N)
`self.b`	Bias parameter	scalar
`self.w`	Weight parameter	$(1 \times D)$
`self.lr`	Learning rate	scalar
`self.l2_reg`	L2 regularization weight	scalar

This forces the magnitude of $\mathbf{w}$ to be small. The trade-off between this and the original cost function is governed by the λ parameter. It is easy to derive a new gradient for $d\ell(\mathbf{x}, \mathbf{y})/d\mathbf{w}$ (left for the reader) using a similar method that we used for the logistic regression model. The gradient for $d\ell(\mathbf{x}, \mathbf{y})/db$ does not change with the added regularization.

Modify the `grad_wrt_w` and `loss` functions to include the term from the new cost function. The λ parameter is represented by the `l2_reg` variable, which to this point has been set to 0. The rest of the functions should not change.

Deliverable: Logistic Regression class

- A working version of logistic_regression.py that passes all of the automated unit tests in `test_logistic_regression.py`.
- Using the same Gaussian dataset as `run_knn.py`, generate a plot of the loss function value vs iteration for learning rates of 0.1, 0.01, and 50.0 for 100 epochs. Use the `plt.legend` function to add a legend to the plot. Save this plot as `log_res_lr.png`.
- A modified working version of `run_logreg.py` that plots the loss function value vs gradient descent iterations on the same Gaussian dataset as `run_knn.py`.

6 Implementing the Support Vector Machine

Now we will implement the support vector machine (SVM) model [5]. Like the logistic regression model, the basic SVM model learns a separating hyper-plane of the form $\mathbf{w}^T\mathbf{x} + b$. The primary difference between the SVM and the logistic regression model is the SVM's hinge loss function. Although there exist more efficient ways to train SVMs [6], we will use the same gradient descent algorithm that we used for logistic regression. The SVM hinge loss is defined as:

$$\ell(\mathbf{x}, \mathbf{y}) = \frac{1}{N}\sum_i \max\left(0, 1 - y_i(\mathbf{w}^T\mathbf{x}_i + b)\right) + \frac{1}{2}\lambda\mathbf{w}^T\mathbf{w} \tag{14}$$

Like logistic regression we will need to compute the gradients $\frac{d\ell(\mathbf{x},\mathbf{y})}{d\mathbf{w}}$ and $\frac{d\ell(\mathbf{x},\mathbf{y})}{db}$. This time we leave the gradient derivation to the reader. Mathematically, the max function does not have a smooth gradient because of the discontinuity at $1 - y_i(\mathbf{w}^T\mathbf{x}_i + b) = 0$. In practice we don't need to worry too much about this discontinuity, and we can just set the derivative at $1 - y_i(\mathbf{w}^T\mathbf{x}_i + b) = 0$ equal to 0.

The project folder includes unit tests in the `test_svm.py`. You will implement an SVM class in the `svm.py`. Similar to KNN and logistic regression, you can create a `run_svm.py` file to test your code with the toy Gaussian dataset. The dimensions of the variables are the same as in the logistic regression model (Table 2). The SVM will use the same gradient descent algorithm defined in 1, however the computation of the gradients will be different. Finally for testing, the SVM does not have a probabilistic output like logistic regression. Instead a value that is less than 0 corresponds to a predicted class of -1 and a value greater than 0 corresponds to a predicted class of +1.

Hint: `ndarray.max` vs `np.maximum`

`nadarray.max` and `np.maximum` are different. Read the documentation of these functions to make sure you are using the appropriate function.

Deliverable: SVM Implementation

- A working version of `svm.py` that passes all of the automated unit tests in `test_svm.py`
- Using the same Gaussian dataset as `run_knn.py`, generate a plot of the loss function value vs gradient descent iteration for learning rates of 0.01, 0.1, and 1 for 100 epochs. Use the `plt.legend` function to add a legend to the plot. Save this plot as `svm_lr.png`.
- A modified working version of `run_svm.py` that plots the loss function value vs gradient descent iteration on the same Gaussian dataset as `run_knn.py`.

7 Using Scikit-Learn Models

There are many proposed models for classification problems. Thankfully instead of implementing them ourselves, there exists high quality Python implementations. The `Scikit-Learn` library provides implementations of many machine learning algorithms with a consistent interface. This means that we can use the same evaluation code and replace one or two lines to use an entirely different machine learning model. In fact, the KNN, SVM and logistic regression classes we implemented in this lab use the same `fit` and `predict` function definitions as in the Scikit-Learn models. So we already know how to use the Scikit-Learn models.

To install Scikit-Learn run the following `pip` command in the terminal:

```
sudo pip install scikit-learn
```

As a first example lets take our previous experiments and replace them with an SVM model from Scikit-Learn. Duplicate the `run_svm.py` file we created earlier and name the new file `run_sklearn_svm.py`. To use Scikit-Learn, first we need to import the SVM models from Scikit-Learn:

```
import sklearn.svm
```

Note that the package for Scikit-Learn is called `sklearn`. The Scikit-Learn SVM model can be instantiated by:

```
model = sklearn.svm.SVC(C=1.0, kernel='linear')
```

Here `C` is a parameter that controls the trade-off between the regularizer and the rest of the cost function. Instead of using a λ to weight the regularization term, the Scikit-Learn implementation weights the $\max(0,\ldots)$ in the cost function by the scalar C. We defined the cost function in terms of λ because this is closer to the optimization literature and similar to the logistic regression example. Note that C and λ are used in similar ways to control the relative weight of the regularizer term. The `kernel='linear'` term will make the SVM implementation use the same $\mathbf{W}^T\mathbf{x} + b$ form that we implemented in our SVM. We will discuss more about the kernel parameter later. Scikit-Learn calls the class `SVC` which stands for Support Vector Classifier. This is to distinguish it from other support vector models that can be used for regression problems.

The data format for the Scikit-Learn models is the same that we have been using for our models. We use the `fit(x,y)` function to train the model and the `predict(y)` function to test the model. The rows of the inputs `x` are samples and the columns are features. `y` is a vector of the corresponding class labels. Run the Scikit-learn SVM on the Gaussian dataset and make sure that the classification accuracy is similar to that of the SVM model you implemented earlier.

Extra Information: Scikit-Learn SVM Implementation

The Scikit-learn implementation (based on Lib-SVM [6]) reformulates the optimization problem as a quadratic programming problem to perform more efficient optimization, so the final parameters will likely be different than the parameters we learned using simple gradient descent. In fact, the Scikit-Learn implementation does not require a learning rate for its optimization procedure. If you are curious, you can benchmark your SVM implementation and compare the training time with the Scikit-Learn implementation.

Non-linear SVM The toy problem we have been using is good to test linear classifiers, however often real world problems cannot be solved with a linear classifier. One example for such a problem is the half-moon problem (Figure 6). Although this problem should be easier than the Gaussian dataset because there are no overlapping data points, the linear classification models we used thus far will not give a good classification accuracy. For this type of problem we need a non-linear classifier.

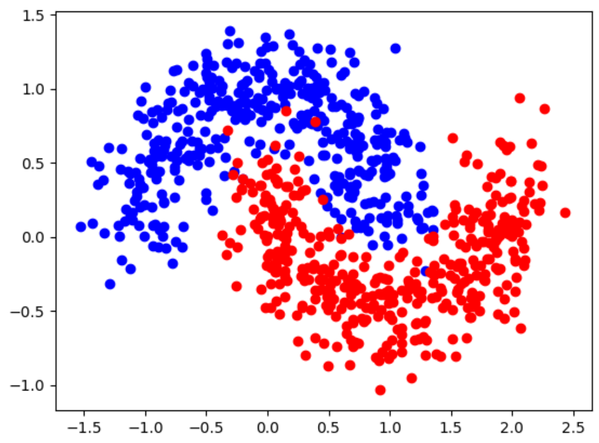


Figure 6: Half moon dataset. The half moon dataset consists of two classes that cannot be linearly separated. This will cause a low classification accuracy for the linear classifiers we implemented in this lab. However if we use a non-linear SVM, we can achieve much better accuracy.

The linear classifier is easier to formulate and optimize, so if possible we would like to keep these advantages. The SVM model uses the so called "kernel trick" to essentially use a linear classifier to classify non-linear data. The kernel trick does this by applying a kernel to transform the input data into more dimensions. The motivation is that in higher dimensions it becomes easier to find a separating hyper-plane because there are more degrees of freedom. Figure 7 shows an intuitive example of this. In one dimension the data cannot be perfectly separated by a linear classifier. However if we apply the kernel $k(x) = x^2$ we can fit a line that perfectly separates the data. A full

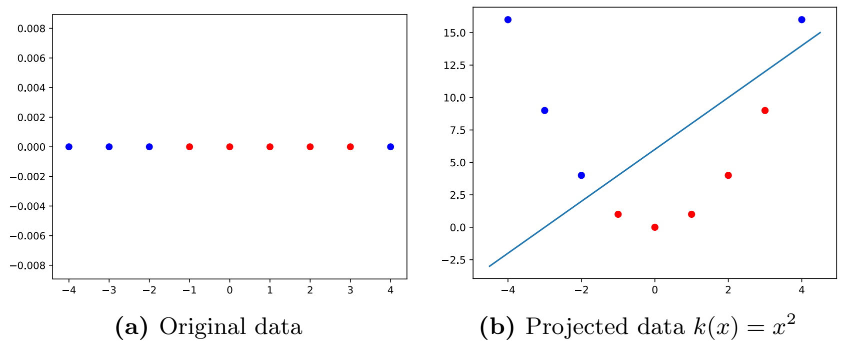


Figure 7: Conceptual example of the SVM kernel trick. (a) shows data with a single feature of two classes: red dots and blue dots. The data is clearly not linearly separable, so a linear SVM will fail with this data. However if we use the function $k(x) = x^2$ to project the data to a second dimension, it is possible to fit a linear separating line to the data. In practice our data often has much more than two dimensions, so it is difficult to visualize the effect of the kernel trick. Nevertheless, this example shows how projecting the data to additional dimensions can help classification.

description of the mathematics of the SVM kernel trick is beyond the scope of this lab, but it can be found in [7].

Scikit-Learn allows 5 different options for the SVM kernel. The `linear` kernel implements the same SVM that we implemented ourselves earlier. `poly` uses a polynomial function for the kernel. `rbf` uses a radial basis (Gaussian) function. `sigmoid` uses a hyperbolic tangent function. Finally `pre-computed` allows for a custom pre-computed kernel.

To demonstrate the non-linear SVM we will use the toy half moon dataset (Figure 6). We have provided a function to generate this dataset in the `datasets.py` file. In a testing script this data can be loaded with the code:

```
x_train, y_train, x_test, y_test = datasets.moon_dataset(n_train=800,
    n_test=800)
```

Extra Information: Scikit-Learn

Scikit-Learn contains implementations of many popular machine learning models, including classification, regression, and unsupervised models. While deep learning can handle many problems, the “shallow” learning models in Scikit-Learn can be used in conjunction with deep learning, or can solve problems on their own. The shallow models can be much more efficient, and could be tried before deep learning when working on a new problem.

You will need to modify `run_sklearn_svm.py` to use this dataset and compare the accuracy using 4 different kernel functions.

Deliverable: Scikit-Learn SVM

- Test the Scikit-Learn linear SVM on the half moon dataset and report the accuracy
- Change the kernel of the Scikit-Learn SVM to polynomial, rbf, sigmoid and report the accuracy. You may need to change other parameters of the kernels to obtain good results.
- A modified working version of `run_sklearn_svm.py` that calculates the accuracy of the Scikit-Learn SVM with different kernels.
- Store these results in a comma separated value file called `svm_results.csv` where each line is of the form `algorithm,0.XXXX`

8 Submission Instructions

We will test your code using unit tests similar to (but not identical to) the unit tests that we used in this lab. Thus you must be sure that your code works for all cases, not just the particular cases in the given unit tests. Do not change any of the names of the classes or functions. Also do not change the folder structure of the code as this is assumed by the testing code used for grading. You will be graded based on how many tests your code passes.

Replace the name of the given source folder with `FIRSTNAME_LASTNAME_LAB2` and zip the folder for submission. The plots that you generated during this lab, as well as the `svm_results.csv` file should be placed in a folder called `results` inside the `FIRSTNAME_LASTNAME_LAB2` folder.

The following are the grading guidelines we will use for this lab:

Table 3: Grading rubric

Points	Description
10	Correct file names and folder structure
10	`ls.py` implementation that passes all tests
5	`ls_error.png` plot of the error of least squares regression vs k
5	`run_ls.py` that plots the error of lease squares regression vs k
10	`knn.py` implementation that passes all tests
5	Plot of KNN accuracy vs k
5	`run_knn.py` that plots KNN accuracy vs k
20	`logistic_regression.py` implementation that passes all tests
5	Plot of logistic regression training loss vs iteration for learning rates 0.1, 0.01 and 50.0
5	`run_logreg.py` that plots logistic regression training loss vs iteration for different lr
20	`svm.py` implementation that passes all tests
5	Plot of SVM training loss vs iteration for learning rates 0.1, 0.01 and 1
5	`run_svm.py` that plots SVM training loss vs iteration for different lr
5	`run_sklearn_svm.py` that runs Scikit-Learn SVM and that calculates the accuracy with different kernels
5	`svm_results.csv` file with results from running Scikit-Learn SVM
120	Total

References

[1] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[2] D. R. Cox, "The regression analysis of binary sequences," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 215–242, 1958.

[3] M. Aly, "Survey on multiclass classification methods," *Neural Networks*, 2005.

[4] T. P. Minka, "A comparison of numerical optimizers for logistic regression," 2003.

[5] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[6] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[7] H. Yu and S. Kim, "SVM tutorial: classification, regression and ranking," in *Handbook of Natural computing*, pp. 479–506, Springer, 2012.