4 Anatomy of a Learning Algorithm

Each learning algorithm has three parts:

- a _____ for one training example, often a function of the difference between estimated and actual label y associated with feature vector $\mathbf{x}$
- a _____ providing an optimization criterion; often an average loss over all training examples
- an _____ that uses training data to satisfy the optimization criterion

e.g. Recall the loss and cost for the algorithms we saw in §3:

Algorithm	**Loss for** $(\mathbf{x}, y)$	**Cost for** $\{(\mathbf{x}_i, y_i)\}_{i=1}^N$
Linear regression	squared error $[f_{\mathbf{w},b}(\mathbf{x}) - y]^2$	$\text{MSE}_{\mathbf{w},b} = \frac{1}{N}\sum_{i=1}^{N}[f_{\mathbf{w},b}(\mathbf{x}_i) - y_i]^2$
Logistic regression	negative log likelihood (regularized) $-\ln\left(f_{\mathbf{w},b}(\mathbf{x}_i)^y\left[1 - f_{\mathbf{w},b}(\mathbf{x})\right]^{1-y}\right)$	$\frac{1}{2}\|\|\mathbf{w}\|\|^2 + C\left[-\sum_{i=1}^{N}\ln\left(f_{\mathbf{w},b}(\mathbf{x}_i)^{y_i}\left[1 - f_{\mathbf{w},b}(\mathbf{x}_i)\right]^{1-y_i}\right)\right]$
SVM	hinge loss (regularized) $\max(0, 1 - y(\mathbf{w}\mathbf{x} + b))$	$\frac{1}{2}\|\|\mathbf{w}\|\|^2 + C\frac{1}{N}\sum_{i=1}^{N}\max(0, 1 - y_i(\mathbf{w}\mathbf{x}_i + b))$
Decision tree[a]	negative log likelihood $-\ln f_{ID3}(\mathbf{x})^y\left[1 - f_{ID3}(\mathbf{x})\right]^{1-y}$	$-\frac{1}{N}\sum_{i=1}^{N}\left[y_i \ln f_{ID3}(\mathbf{x}_i) + (1 - y_i)\ln\left(1 - f_{ID3}(\mathbf{x}_i)\right)\right]$
kNN[b]		

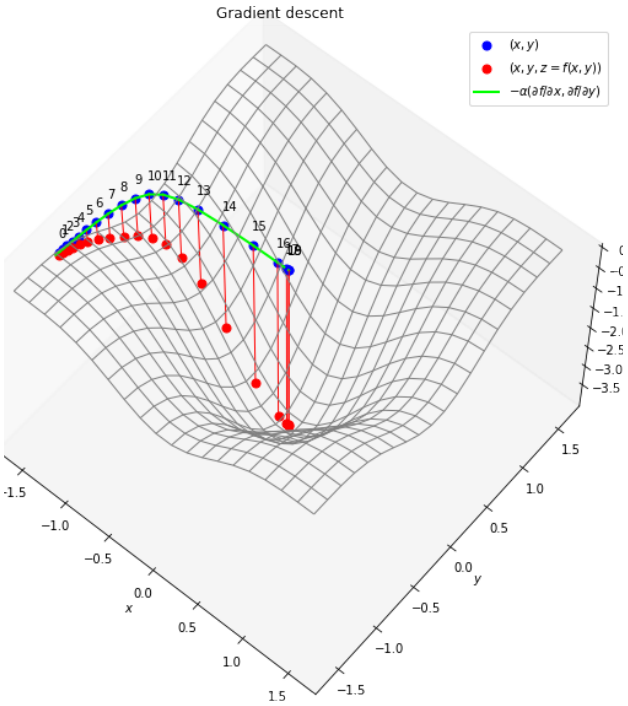
[a] Recall: Burkov says ID3 approximately minimizes $-\ln L_{\mathbf{w},b}$.
[b] kNN does not easily fit this table.

When we do not have a closed-form solution for minimizing the cost, we use a *numerical optimization* method like gradient descent (below).[1]

[1] We want to minimize $\text{MSE}_{\mathbf{w},b}$ or $-\ln L_{\mathbf{w},b}$ or SVM's cost function over $\mathbf{w}$ and b. However, I present gradient descent from a mathematical and graphical perspective in which we minimize over $\mathbf{x}$ (which in this context does not refer to our feature vector) or over (x, y). Alas, I haven't figured out effortless notation.

Gradient Descent

Gradient descent iteratively steps _____ the direction of and proportional to the _____ of the gradient of a function to seek a *local* minimum.



Recall: For differentiable $z = f(\mathbf{x})$, where $\mathbf{x} = (x_1, \ldots, x_n)$, the the *gradient* of f is $\nabla f(x_1, \ldots, x_n) = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right)$. To minimize f by *gradient descent*, choose an initial point $(x_1, \ldots, x_n)$ and iteratively move opposite the gradient by iterating on

$$\mathbf{x}_{i+1} = \_\_\_\_\_$$

where α is the *learning rate* controlling step size.

Many improvements are possible. e.g.

- Decrease α at each step.
- Set α locally optimally at each step via a *line search*, guaranteeing convergence for a well-behaved f.

Starting at a random point, gradient descent finds a *local* minimum of f.

$z = f(\mathbf{x})$ is *convex* if the line segment between any two points on its graph is not below the graph. If f is convex,[2] gradient descent can find its *global* minimum.

e.g. Minimize $f(x) = x^2$ starting at $x = -2$ with $\alpha = 1$. Repeat with $\alpha = \frac{3}{4}$.

e.g. Run gradient descent with $\alpha = 0.4$ to minimize $z = f(x, y) = x^2 + y^2 - 6x - 4y + 13$. Start at $(0, 0)$ and find the next two points on the descent path.

$\nabla f(x, y) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right) = (\quad\quad, \quad\quad)$

i	$\mathbf{x}_i = (x_i, y_i)$	$\nabla f(x_i, y_i)$	$-\alpha \nabla f(x_i, y_i)$
0	$(0, 0)$		
1			
2			

[2] It will find the global minimum if additionally ∇f is *Lipschitz continuous* and γ is chosen by a good line search.

Stochastic Gradient Descent

We can use gradient descent (or a variant) to minimize the cost functions for linear regression, logistic regression, and a support vector machine.

Note the average loss "$\frac{1}{N}\sum_{i=1}^{N}\ldots$" over the N training examples in these cost functions:

- linear regression: minimize $\text{MSE}_{\mathbf{w},b} = \dfrac{1}{N}\displaystyle\sum_{i=1}^{N}\left[f_{\mathbf{w},b}(\mathbf{x}_i) - y_i\right]^2$
- logistic regression: minimize $-\ln L_{\mathbf{w},b} = \dfrac{1}{2}||\mathbf{w}||^2 + C(-N)\left[\dfrac{1}{N}\displaystyle\sum_{i=1}^{N}\ln\left(f_{\mathbf{w},b}(\mathbf{x}_i)^{y_i}\left[1 - f_{\mathbf{w},b}(\mathbf{x}_i)\right]^{1-y_i}\right)\right]$
- SVM: minimize $\dfrac{1}{2}||\mathbf{w}||^2 + C\dfrac{1}{N}\displaystyle\sum_{i=1}^{N}\max(0, 1 - y_i(\mathbf{w}\mathbf{x}_i + b))$

That average persists when we find the gradient with respect to each element of $\mathbf{w}$ and b. Evaluating it over a large number N of (possibly high-D) examples is computationally expensive.

Stochastic Gradient Descent (SGD) approximates gradient descent by evaluating the average loss not over _____ examples but rather over _____ for speed.[3]

Consider SGD algorithms when the regular ones are slow on your data set.

Python

- `from sklearn.linear_model import SGDClassifier`
  - `clf = SGDClassifier(loss='hinge', penalty='l2', alpha=0.0001, max_iter=1000)`:
    - `loss` options include `'hinge'` for a linear SVM, `'log_loss'` for logistic regression
    - `penalty` (regularization term) options include `'l2'` and `'l1'`
    - `alpha` is a constant multiplying the regularization term (related to our C)
    - `max_iter` is the maximum number of passes over the training data
- `from sklearn.linear_model import SGDRegressor`
  - `model = SGDRegressor(loss='squared_error', penalty='l2', alpha=0.0001, max_iter=1000)` gives SGD OLS regression, whose regularization we will see in §05
- User guide: `https://scikit-learn.org/stable/modules/sgd.html`
- Reference manual:
  - classification:
    `https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html`
  - regression:
    `https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDRegressor.html`

[3] A variant uses a *mini-batch* of several randomly-selected examples.

Code pattern for using scikit-learn

- `import ...` loads a module
- `ml = <classifier or regressor>(...)` gets a classifier or regressor and sets its hyperparameters; e.g.
  - `ml = svm.SVC(kernel='linear', C=1)`
  - `ml = svm.SVC(kernel='rbf', C=1, gamma='scale')`
  - `ml = linear_model.LinearRegression()`
  - `ml = linear_model.LogisticRegression(C=1)`
  - `ml = DecisionTreeClassifier(criterion='entropy', max_depth=None, min_impurity_decrease=0)`
    `ml = DecisionTreeRegressor(criterion='squared_error', max_depth=None)`
  - `ml = kNeighborsClassifier(n_neighbors=5, weights='uniform')`
  - `ml = kNeighborsRegressor(n_neighbors=5)`
- `ml.fit(X, y)` runs the training algorithm
- `ml.coef_` gives $\mathbf{w}^*$ and `ml.intercept_` gives b^* (for SVM and linear & logistic regression)
- `ml.predict(X)` does classification or regression
- `ml.predict_proba(X)` gives classification outcome probabilities for examples in `X` (for logistic regression, decision tree, kNN and, optionally, SVM)
- `ml.score(X, y)` gives accuracy on `X` with respect to `y` or R^2 or another performance measure

Details of Learning Algorithms

- Different algorithms have different hyperparameters; e.g.
  - SVM: _____ for regularization; _____ kernel coefficient for `kernel='rbf'`
  - logistic regression: _____ for regularization
  - ID3 decision tree: $d =$ _____ and $\epsilon =$ _____
  - kNN: _____, choice of _____ , choice of weights (uniform by default)
  - gradient descent: learning rate _____
- Some algorithms, including decision trees, accept categorical features like "color" taking values like "red" and "blue". Some require numbers. Scikit-learn uniformly uses _____ features. We will see in §5 how to map categories to numbers.
- Some algorithms, including SVM, allow us to weight each class. Weighting a class higher discourages training errors for that class.
- Some algorithms, including SVM, decision tree, and kNN, can be used for classification and regression. Others address only one task.