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 $x$
- 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:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Loss for $(x, y)$</th>
<th>Cost for ${(x_i, y_i)}_{i=1}^N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear regression</td>
<td>$[(f_{w,b}(x)) - y]^2$</td>
<td>$\text{MSE}<em>{w,b} = \frac{1}{N} \sum</em>{i=1}^N [(f_{w,b}(x_i)) - y_i]^2$</td>
</tr>
<tr>
<td>Logistic regression</td>
<td>negative log likelihood (regularized)</td>
<td>$\frac{1}{2}</td>
</tr>
<tr>
<td>SVM</td>
<td>hinge loss (regularized)</td>
<td>$\frac{1}{2}</td>
</tr>
<tr>
<td>Decision tree</td>
<td>negative log likelihood (regularized)</td>
<td>$\frac{1}{N} \sum_{i=1}^N \left[y_i \ln f_{ID3}(x_i) + (1 - y_i) \ln (1 - f_{ID3}(x_i)) \right]$</td>
</tr>
<tr>
<td>kNN $^b$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$^a$Recall: Burkov says ID3 approximately minimizes $-\ln L_{w,b}$.

$kNN$ 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 $\text{MSE}_{w,b}$ or $-\ln L_{w,b}$ or SVM’s cost function over $w$ and $b$. However, I present gradient descent from a mathematical and graphical perspective in which we minimize over $x$ (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 \( z = f(x) \), where \( x = (x_1, \ldots, x_n) \), 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

\[ x_{i+1} = \________________________ \]

where \( \alpha \) is the learning rate controlling step size.

Many improvements are possible. e.g.

- Decrease \( \alpha \) at each step.
- Set \( \alpha \) 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 \( f \).

\( z = f(x) \) 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.

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

<table>
<thead>
<tr>
<th>( i )</th>
<th>( x_i = (x_i, y_i) )</th>
<th>( \nabla f(x_i, y_i) )</th>
<th>(-\alpha \nabla f(x_i, y_i))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>((0,0))</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^2\text{It will find the global minimum if additionally } \nabla f \text{ is Lipschitz continuous and } \gamma \text{ is chosen by a good line search.}\)
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 \( \text{MSE}_{w,b} = \frac{1}{N} \sum_{i=1}^{N} [f_{w,b}(x_i) - y_i]^2 \)
- logistic regression: minimize \( -\ln L_{w,b} = \frac{1}{2} ||w||^2 + C(-N) \left[ \frac{1}{N} \sum_{i=1}^{N} \ln \left( f_{w,b}(x_i)y_i [1 - f_{w,b}(x_i)]^{1-y_i} \right) \right] \)
- SVM: minimize \( \frac{1}{2} ||w||^2 + C \frac{1}{N} \sum_{i=1}^{N} \max(0, 1 - y_i (wx_i + b)) \)

That average persists when we find the gradient with respect to each element of \( w \) 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 \( \ldots \) examples but rather over \( \ldots \) for speed.

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


**Reference manual:**


---

\(^3\)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 $w^*$ 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 $\epsilon =$ _________________
  - $k$NN: ______, 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 $k$NN, can be used for classification and regression. Others address only one task.