Lesson 07
Notes on Chapter 4

CSC357 Advanced Topics—Machine Learning

24 January 2020

- often represent vectors as column vectors
- column vectors are 2D arrays with a single column
- if θ and $\mathbf{x}$ are column vectors then prediction is...
  - $\hat{y} = \theta^T \mathbf{x}$, where θ^T
  - and it is the transpose of θ (a row vector instead of a column vector)
  - $\theta^T \mathbf{x}$ is the matrix multiplication of θ^T and $\mathbf{x}$.
  - same prediction, represented as a single-cell matrix rather than a scalar value
  - use this notation to avoid switching between dot products and matrix multiplications
- how to train Linear Regression model?
- training means setting parameters to get best fit to training set
- most common performance measure is Root Mean Square Error (RMSE)
- find value of θ that minimizes the RMSE
- simpler to minimize the mean squared error (MSE) than the RMSE
- (value that minimizes a function also minimizes its square root)
- MSE cost function for a Linear Regression model...

$$MSE(\mathbf{X}, h_\theta = \frac{1}{m} \sum_{i=1}^{m} (\theta^T \mathbf{x}^{(i)} - y^{(i)})^2$$

- we write h_θ instead of just h to clear that model is parametrized by the vector θ

- to simplify, just write $MSE(\theta)$ instead of $MSE(\mathbf{X}, h_\theta)$
- *closed-form* solution (to the problem of training a Linear Regression model) is called the *Normal Equation*

$$\hat{\theta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

- in this equation...
  - $\hat{\theta}$ is the value of θ that minimizes cost function
  - $\mathbf{y}$ is the vector of target values containing $y^{(i)}$ to $y^{(m)}$
- generate some linear-looking data to test this equation...
  - function to generate the data is $y = 4 + 3x_1 1+$ Gaussian noise

```
import numpy as np

X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

- compute $\hat{\theta}$ using Normal Equation...
  - use NumPy's inv() function (from np.linalg module) to compute matrix inverse
  - dot() method for matrix multiplication:

```
# add x0 = 1 to each instance
X_b = np.c_[np.ones((100, 1)), X]
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

- see what the equation found—

```
theta_best
array([[4.21509616],          [2.77011339]])
```

- hoped for $\theta_0 = 4$ and $\theta_1 = 3$ (instead of $\theta_0 = 4.215$ and $\theta_1 = 2.770$)
- close enough! (noise accounts for difference)
- now make predictions using $\hat{\theta}$...

```
X_new = np.array([[0], [2]])
# add x0 = 1 to each instance
X_new_b = np.c_[np.ones((2, 1)), X_new]
y_predict = X_new_b.dot(theta_best)
y_predict
array([[4.21509616],
       [9.75532293]])
```

- plot this models predictions...

```
plt.plot(X_new, y_predict, "r-")
plt.plot(X, y, "b.")
plt.axis([0, 2, 0, 15])
plt.show()
```

- performing Linear Regression using Scikit-Learn is simple...

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(X, y)
lin_reg.intercept_, lin_reg.coef_
        (array([4.21509616]), array([[2.77011339]]))
lin_reg.predict(X_new)
array([[4.21509616],
       [9.75532293]])
```

- LinearRegression class based on scipy.linalg.lstsq()
- scipy.linalg.lstsq() stands for "least squares"
- can call scipy.linalg.lstsq() directly...

```
theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_b, y,
        rcond=1e-6)
theta_best_svd
array([[4.21509616],
       [2.77011339]])
```

- function above computes $\hat{\theta} = \mathbf{X}^{+}\mathbf{y}$
  - $\mathbf{X}^{+}$ is *pseudoinverse* of $\mathbf{X}$
  - (specifically, the Moore-Penrose inverse)
- can compute pseudoinverse directly with np.linalg.pinv()

```
np.linalg.pinv(X_b).dot(y)
array([[4.21509616],
       [2.77011339]])
```

- pseudoinverse computed using a standard matrix factorization technique
  - technique is *Singular Value Decomposition* (SVD)
  - decomposes training set matrix $\mathbf{X}$ into product of 3 matrices
  - here's the product: $\mathbf{U\Sigma V}^{T}$
  - see numpy.linalg.svd()

  - pseudoinverse is computed as $\mathbf{X}^+ = \mathbf{V}\boldsymbol{\Sigma}^+\mathbf{U}^T$
  - to compute matrix $\boldsymbol{\Sigma}^+$
    * algorithm takes $\boldsymbol{\Sigma}$ and sets to zero all values $<$ tiny threshold value,
    * then replaces all nonzero values with their inverse
    * finally transposes the resulting matrix
- approach more efficient than computing Normal Equation
- plus, handles edge cases nicely
- indeed, the Normal Equation may not work if matrix $\mathbf{X}^T\mathbf{X}$ is not invertible (i.e., singular), such as if $m < n$ or if some features are redundant
- Normal Equation computes inverse of $\mathbf{X}^T\mathbf{X}$
- that is a $(n+1) \times (n+1)$ matrix
- (n is the number of features)
- computational complexity of inverting such a matrix typically about $O(n^{2.4})$ to $O(n^3)$
- (doubling number of features multiplies computation time by roughly $2^{2.4} = 5.3$ to $2^3 = 8$)
- SVD approach used by Scikit-Learns LinearRegression class is about $O(n^2)$
- (doubling number of features multiplies computation time by roughly 4)
- both the Normal Equation and SVD approach very slow when number of features grows large (e.g., $100,000$)
- on positive side, both linear with regard to number of instances in training set (they are $O(m)$)
- so they handle large training sets efficiently (provided they can fit in memory)
- once trained, Linear Regression model predictions are very fast
- computational complexity linear with regard to both number of instances (on which to make predictions) and number of features
- $2\times$ as many predictions (or $2\times$ features) takes $2\times$ time