On Machine Learning

Alapan Chaudhuri

November 2020

1 On Machine Learning

For codes and more please refer to: **this**

- **Supervised Learning**: data is labeled and the program learns to predict the output from the input data
- **Unsupervised Learning**: data is unlabeled and the program learns to recognize the inherent structure in the input data

1.1 Scikit-Learn Models

1.1.1 Linear Regression

Import and create the model:

```
from sklearn.linear_model import LinearRegression
your_model = LinearRegression()
```

Fit: `your_model.fit(x_training_data, y_training_data)` - `.coef_`: contains the coefficients - `.intercept_`: contains the intercept

Predict: `predictions = your_model.predict(your_x_data)` - `.score()`: returns the coefficient of determination R^2

1.1.2 Naive Bayes

Import and create the model:

```
from sklearn.naive_bayes import MultinomialNB
your_model = MultinomialNB()
```

Fit: `your_model.fit(x_training_data, y_training_data)`

Predict:

```
# Returns a list of predicted classes - one prediction for every data
    point
predictions = your_model.predict(your_x_data)

# For every data point, returns a list of probabilities of each class
probabilities = your_model.predict_proba(your_x_data)
```

1.1.3 K-Nearest Neighbors

Import and create the model:

```
from sklearn.neigbors import KNeighborsClassifier
your_model = KNeighborsClassifier(n_neighbors = 5)
```

Fit:

```
your_model.fit(x_training_data, y_training_data)
```

Predict:

```
# Returns a list of predicted classes - one prediction for every data
    point
predictions = your_model.predict(your_x_data)

# For every data point, returns a list of probabilities of each class
probabilities = your_model.predict_proba(your_x_data)
```

1.1.4 K-Means

Import and create the model:

```
from sklearn.cluster import KMeans
your_model = KMeans(n_clusters=4, init='random')
```

- `n_clusters`: number of clusters to form and number of centroids to generate
- `init`: method for initialization
  - `k-means++`: K-Means++ [default]
  - `random`: K-Means

- `random_state`: the seed used by the random number generator [optional]

Fit:

```
your_model.fit(x_training_data)
```

Predict:

```
predictions = your_model.predict(your_x_data)
```

1.1.5 Validating the Model

Import and print accuracy, recall, precision, and F1 score:

```
from sklearn.metrics import accuracy_score, recall_score,
    precision_score, f1_score

print(accuracy_score(true_labels, guesses))
print(recall_score(true_labels, guesses))
print(precision_score(true_labels, guesses))
print(f1_score(true_labels, guesses))
```

Import and print the confusion matrix:

```
from sklearn.metrics import confusion_matrix

print(confusion_matrix(true_labels, guesses))
```

1.1.6 Training Sets and Test Sets

```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(x, y,
    train_size=0.8, test_size=0.2)
```

- `train_size`: the proportion of the dataset to include in the train split
- `test_size`: the proportion of the dataset to include in the test split
- `random_state`: the seed used by the random number generator

2 Linear Regression

2.0.1 Equations Associated:

- Equation for fit

$$y = mx + c$$

- Equation for loss calculation

$$\sum_{\forall i \in n} (y_{actual} - y_{fit})^2$$

- Equation for Gradient Descent for intercept

$$b_{gradient} = \frac{2}{n} \sum_{\forall i \in n} (mx_i + b - y_i)$$

- Equation for Gradient Descent for slope

$$m_{gradient} = \frac{2}{n} \sum_{\forall i \in n} x_i(mx_i + b - y_i)$$

- Equation for b_{new} and m_{new} are

$$b_{new} = b - (\alpha \times b_{gradient})$$

$$m_{new} = m - (\alpha \times m_{gradient})$$

2.0.2 Complete Code Construction

```
def get_gradient_at_b(x, y, b, m):
  diff = 0
  n = len(x)
  for i in range(0, n):
    val = m*x[i]+b - y[i]
    diff += val
  b_gradient = 2*diff/n
  return b_gradient

def get_gradient_at_m(x, y, b, m):
  diff = 0
  n = len(x)
  for i in range(0, n):
    val = m*x[i]+b - y[i]
    diff += val*x[i]
  m_gradient = 2*diff/n
  return m_gradient
```

```
def step_gradient(b_current, m_current, x, y, learning_rate):
    b_gradient = get_gradient_at_b(x, y, b_current, m_current)
    m_gradient = get_gradient_at_m(x, y, b_current, m_current)
    b = b_current - (learning_rate * b_gradient)
    m = m_current - (learning_rate * m_gradient)
    return [b, m]


def gradient_descent(x, y, learning_rate, num_iterations):
  b, m = 0, 0
  for i in range(0, num_iterations):
    b, m = step_gradient(b, m, x, y, learning_rate)
  return [b, m]

X_values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
Y_values = [52, 74, 79, 95, 115, 110, 129, 126, 147, 146, 156, 184]

b, m = gradient_descent(X_values, Y_values, 0.01, 1000)

y = [m*x + b for x in X_values]

plt.plot(X_values, Y_values, "o")
plt.plot(X_values, y)

plt.show()
```

2.0.3 Multiple Linear Regression

The equation for fit is $y = c + m_1x_1 + ... + m_nx_n$.

2.0.4 Throwback on Linear Regression (analytical approach)

As Zeeshan said, Loss Function is a social construct, we have out current loss function as $C = \frac{\sum(y_i-(mx_i+b))^2}{n}$.

Now let us figure out the gradient descent with respect to b,

$$\frac{\partial C}{\partial b} = \frac{2}{n}\sum_{i=1}^{n}((y_i - (mx_i + b)))(-1) = \nabla_b C$$

which shows the direction to go for changing b inorder to increase the Loss.

Thus, required direction for reducing Loss shall be $-\nabla_b C$ and the required equation for changing the value of b shall be

$$b = b - \nabla_b C db$$

.

3 Classification

3.0.1 Distances

1. Eulclidean Distance:
$$d = \sqrt{\sum_{\forall i \in n} (A_i - B_i)^2}$$
2. Cityblock/Manhattan Distance:
$$d = \sum_{\forall i \in n} |A_i - B_i|$$
3. Hamming Distance: d_n where $d_0 = 0$ and,
$$d_i = \begin{cases} d_{i-1} & \text{when } A_i = B_i \\ d_{i-1} + 1 & \text{else} \end{cases}$$

3.0.2 Normalisation

- Min-max normalization: Guarantees all features will have the exact same scale but does not handle outliers well.
$$val = \frac{val - min}{max - min}$$
- Z-score normalization: Handles outliers, but does not produce normalized data with the exact same scale.
$$val = \frac{val - \mu}{\sigma}$$

3.0.3 Supervised ML and Classification

Supervised machine learning algorithms are amazing tools capable of making predictions and classifications.

In order to test the effectiveness of your algorithm, we'll split this data into:

- training set
- validation set
- test set

3.0.4 K-Nearest Neighbor Algorithm

```
1. Normalize the data
2. Find the k nearest neighbors
3. Classify the new point based on those neighbors
```

```
 # The Classification Function
   def classify(unknown, dataset, labels, k):
       distances = []
       for title in dataset:
           movie = dataset[title]
           distance_to_point = dist.euclidean(movie, unknown)
           distances.append([distance_to_point, title])
       distances.sort()

       neighbors = distances[0:k]
       num_good, num_bad = 0, 0
       for movie in neighbors:
           title = movie[1]
           if labels[title] == 0:
               num_bad += 1
           else:
               num_good +=1
       return 1 if num_good > num_bad else 0


 # The Validation Error Finder
   def find_validation_accuracy(training_set, training_labels,
       validation_set, validation_labels, k):
       num_correct = 0.0

       for movie in validation_set:
           guess = classify(validation_set[movie], training_set,
               training_labels, k)
           if guess == validation_labels[movie]:
               num_correct += 1

       validation_error = num_correct/len(validation_set)
       return validation_error
```

3.0.5 K-Nearest Neighbor Regression

Core difference: instead of counting the number of good and bad neighbors, the regressor averages their "label-value" (for e.g., IMDb rating).

```
   from scipy.spatial import distance as D
```

```python
def predict(unknown, dataset, movie_ratings, k):
  distances = []
  for title in dataset:
    movie = dataset[title]
    distance_to_point = D.euclidean(movie, unknown)
    distances.append([distance_to_point, title])
  distances.sort()

  neighbors = distances[0:k]
  avg = 0.0
  for movie in neighbors:
    avg += movie_ratings[movie[1]]
  avg /= k
  return avg
```

We can also use **weighted average** for the regression.

```python
from scipy.spatial import distance as D

def predict(unknown, dataset, movie_ratings, k):
  distances = []
  for title in dataset:
    movie = dataset[title]
    distance_to_point = D.euclidean(movie, unknown)
    distances.append([distance_to_point, title])
  distances.sort()

  neighbors = distances[0:k]
  numerator, denominator = 0.0, 0.0
  for movie in neighbors:
    numerator += movie_ratings[movie[1]]/movie[0]
    denominator += 1/movie[0]
  avg = numerator/denominator
  return avg
```

3.1 Accuracy and F_1 score

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{False Positives} + \text{False Negatives} + \text{True Positives} + \text{True Negatives}}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

$$F_1 \text{ score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

4 Logistic Regression

$$\text{Logistic Odds, log-odds} = \log \frac{x}{1-x}, \text{ where } x = P(\text{event occurring})$$

$$\text{Sigmoid Function, } h(x) = \frac{1}{1+e^{-z}} \text{ where } z = \text{log-odds}$$

4.1 Mathematical Derivation

Look at these handwritten notes for more info (I am not sure if that is required).

Consider $X = b_0 + b_1 x_1 + ... + b_n x_n$ to be a linear regression model, now it is pretty clear that this can't be used to perform a binary classification since the results of a mutli-linear function can't be mapped between $[0, 1]$.

Now, let us transform the results of the the above function using a logistic function, i.e. $\frac{C}{1+e^{X_0 - X}}$, a simplification of which is the sigmoid function as stated earlier.

Now, let $P(X) = h(X)$, then

$$P(X) = \frac{1}{1+e^{-X}}$$
$$\implies \frac{P(X)}{1-P(X)} = e^X$$
$$\implies odds(X) = e^X$$
$$\implies \log odds(X) = X = b_0 + b_1 x_1 + ... + b_n x_n$$

Thus, log-odds is equivalent to the multi-linear function formulated. So, now we have broken the entire classification problem into finding the optimal values for the coefficients b_i.

Now, to solve this problem we can't use regular MSE as the Cost/Loss function because of th possible presence of multiple local minimas.

So, now the only job left for us to do is to figure out the suitable Cost function. And that will be Cross-Entropy.

Consider a case of binary classification, now let: - $\rho(x)$ = expected probability (outcome) of a data point (as an event) - $r(x)$ = predicted probabilty according to model (outcome) of a single data point (event) - Required cross entropy for the above probability distributions = $H(\rho, r)$

Now we have,

$$H(\rho, r) = E_{x\sim\rho} - \log(r(x))$$

Thus, if we have a binary classification problem (over a single data-point),

$$H(\rho, r) = \sum_{x\in\{0,1\}} -\rho(x)\log(r(x))$$

$$\implies H(\rho, r) = -[\rho(0)\log(r(0)) + \rho(1)\log(r(1))]$$

$$\implies H(\rho, r) = -[y\log(h(X)) + (1-y)(\log(1-h(X)))]$$

Thus, the required cost function over multiple (m) data-points,

$$C = -\frac{1}{m}\sum_{i\in m}[y^{(i)}\log(h(X^{(i)})) + (1-y^{(i)})(\log(1-h(X^{(i)})))]$$

5 Multi-Class Classification

Model is represented by, $a(x) = arg(max_{k\in\{1,2,\ldots,K\}}\omega_k^T x)$

5.0.1 The Loss Function

$$L(\omega, b) = -\sum_{i=1}^{l}\sum_{k=1}^{K}[y_i = K]\log\frac{e^{\omega_k^T x_i}}{\sum_{j=1}^{K}\omega_j^T x_i}$$

6 Optimisation

6.1 Managing Overfitting

- Segregate a certain holdout set out of the net training data available
- Cross-validation