Machine learning Algorithms

... Algorithm

```
...rt numpy as np

def euclidean-distance(x1, x2):
    return np.sqrt(np.sum((x1-x2)**2))

class KMeans:
    def __init__(self, K=3, max-iters=100):

        self.K = K
        self.max-iters = max-iters

        # list of sample indices for each cluster
        self.clusters = [[] for _ in range(self.K)]

        # mean feature vector for each cluster
        self.centroids = []

    def predict(self, X):
        self.X = X
        self.n-samples, self.n-features = X.shape

        # initialize centroids
        random-sample-idxs = np.random.choice(self.n-samples, self.K, replace=
                                                False)
        self.centroids = [self.X[idx] for idx in random-sample-idxs]
```

```
# optimization
for i in range (self.max_iters):
    # update clusters
    self.clusters = self._create_clusters (self.centroids)

    # update centroids
    centroids_old = self.centroids
    self.centroids = self._get_centroids (self.clusters)

    # check if converged
    if self._is_converged (centroids_old, self.centroids):
        break

    # return cluster labels
    return self._get_cluster_labels (self.clusters)

def _get_cluster_labels (self, clusters):
    labels = np.empty (self.n_samples)
    for cluster_idx, cluster in enumerate (clusters):
        for sample_idx in cluster:
            labels [sample_idx] = cluster_idx

    return labels.
```

```
def _create_clusters(self, centroids):
    clusters = [[] for _ in range(self.K)]
    for idx, sample in enumerate(self.X):
        centroid_idx = self._closest_centroid(sample, centroids)
        clusters[centroid_idx].append(idx)
    return clusters.

def _closest_centroid(self, sample, centroids):
    distances = [euclidean_distance(sample, point) for point in centroids]
    closest_idx = np.argmin(distances)
    return closest_idx

def _get_centroids(self, clusters):
    centroids = np.zeros((self.K, self.n_features))
    for cluster_idx, cluster in enumerate(clusters):
        cluster_mean = np.mean(self.X[cluster], axis=0)
        centroids[cluster_idx] = cluster_mean
    return centroids.

def _is_converged(self, centroids_old, centroids):
    distances = [euclidean_distance(centroids_old[i], centroids[i]) for i in
                 range(self.K)]
    return sum(distances) == 0.
```

k-nearest neighbors Algorithm

```
import math.

# Euclidean distance calculation
def euclideanDistance ( instance1, instance2, length):
    distance = 0
    for x in range (length):
        distance += math.pow ( instance1[x] - instance2[x], 2)
return math.sqrt (distance)

# Neighbors : selecting subset with the smallest distance
def getNeighbors ( trainingSet , testInstance, k):
    distances = []
    length = len ( testInstance) -1
    for x in range ( len ( trainingSet)):
        dist = euclideanDistance ( testInstance, trainingSet [x], length)
        distances. append ( ( trainingSet [x], dist))
    distances. sort ( key = operator. itemgetter (1))
    neighbors = []
    for x in range (k):
        neighbors. append ( distances [x][0])
return neighbors.
```

```
def getResponse(neighbors):
    classVotes = {}
    for x in range(len(neighbors)):
        response = neighbors[x][-1]
        if response in classVotes:
            classVotes[response] += 1
        else:
            classVotes[response] = 1
    SortedVotes = sorted(classVotes.items(), key = operator.itemgetter(1),
                         reverse = True)
    return SortedVotes[0][0]

def getAccuracy(testSet, predictions):
    correct = 0
    for x in range(len(testSet)):
        if testSet[x][-1] in predictions[x]:
            correct = correct + 1
    return (correct / float(len(testSet)) * 100)

def main():
    predictions = []
    k = 5
    for x in range(len(testSet)):
        neighbors = getNeighbors(trainingSet, testSet[x], k)
        result = getResponse(neighbors)
        predictions.append(result)
    accuracy = getAccuracy(testSet, predictions)
```

(5) KNN Algorithm

1. Load the data
2. Initialize K to the chosen number of neighbors
3. For each example in the data
   (a) Calculate the distance b/w the query example and the current example from the data
   (b) Add the distance and the index of the example to an ordered collection.
4. Sort the ordered collection of distances and indices from smallest to largest (in ascending order) by the distances
5. Pick the first K entries from the sorted collection
6. Get the labels of the selected K entries
7. If regression, return the mean of the K labels
8. If classification, return the mode of the K labels.

Linear Regression Algorithm

```
import numpy as np

class LinearRegression:

    def __init__(self, learning_rate=0.001, n_iters=1000):
        self.lr = learning_rate
        self.n_iters = n_iters
        self.weights = None
        self.bias = None

    def fit(self, X, y):
        n_samples, n_features = X.shape

        # initialize parameters
        self.weights = np.zeros(n_features)
        self.bias = 0

        # gradient descent
        for _ in range(self.n_iters):
            y_predicted = np.dot(X, self.weights) + self.bias

            # compute gradients
            dw = (1/n_samples) * np.dot(X.T, (y_predicted - y))
            db = (1/n_samples) * np.sum(y_predicted - y)

            # update parameters
            self.weights -= self.lr * dw
            self.bias -= self.lr * db
```

```
def mean_squared_error(y_true, y_pred):
    return np.mean((y_true - y_pred)**
```

```
    def predict(self, X):
        y_approximated = np.dot(X, self.weights) + self.bias
        return y_approximated

regressor = LinearRegression(learning_rate=0.01, n_iters=1000)
regressor.fit(X_train, y_train)
predictions = regressor.predict(X_test)

mse = mean_squared_error(y_test, predictions)
```

Logistic Regression Algorithm

```
import numpy as np

class LogisticRegression:

    def __init__(self, lr=0.001, epochs=1000):
        self.lr = lr
        self.epochs = epochs

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def cost_function(self, X, y, weights):
        z = np.dot(X, weights)
        predict_1 = y * np.log(self.sigmoid(z))
        predict_0 = (1-y) * np.log(1 - self.sigmoid(z))
        return -np.sum(predict_1 + predict_0) / X.shape[0]

    def fit(self, X, y):
        loss = []
        weights = np.random.rand(X.shape[1])
        for _ in range(self.epochs):
            y_hat = self.sigmoid(np.dot(X, weights))
            weights -= self.lr * np.dot(X.T, y_hat - y) / X.shape[0]
            loss.append(self.cost_function(X, y, weights))

    def accuracy(y_true,
        accuracy = np.sum(
        return accuracy
```

```
def predict (self, X):
    z = np.dot (X, self.weights)
    return np.array ( [1 if i > 0.5 else 0 for i in self.sigmoid (z)])

regressor = LogisticRegression ()
regressor.fit (X_train, y_train)
predictions = regressor.predict (X_test)
accuracy (y_test, predictions).
```

SVM Algorithm

```
import numpy as np

class SVM:
    def __init__(self, learning_rate = 0.001, lambda_param = 0.01, n_iters = 100
        self.lr = learning_rate
        self.lambda_param = lambda_param
        self.n_iters = n_iters
        self.w = None
        self.b = None

    def fit(self, X, y):
        y_ = np.where(y<=0, -1, 1)
        n_samples, n_features = X.shape

        self.w = np.zeros(n_features)
        self.b = 0.

        for _ in range(self.n_iters):
            for idx, x_i in enumerate(X):
                condition = y_[idx] * (np.dot(x_i, self.w) - self.b) >= 1
                if condition:
                    self.w -= self.lr * (2 * self.lambda_param * self.w)
                else:
                    self.w -= self.lr * (2 * self.lambda_param * self.w -
                    self.b -= self.lr * y_[idx]                 np.dot(x_i, y_[idx]))
```

self.weights, self.loss = weights, loss

```
def predict (self, X):
    linear_output = np.dot (X, self.w) - self.b
    return (np.sign (linear_output))

clf = SVM ()
clf.fit (X, y)
predictions = clf.predict (X)
```

Naive Bayes Algorithm

```
class NaiveBayes:

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self._classes = np.unique(y)
        n_classes = len(self._classes)

        # initialize mean, variance, priors
        self._mean = np.zeros((n_classes, n_features), dtype=np.float64)
        self._var = np.zeros((n_classes, n_features), dtype=np.float64)
        self._priors = np.zeros(n_classes, dtype=np.float64)

        for c in self._classes:
            X_c = X[c == y]
            self._mean[c, :] = X_c.mean(axis=0)
            self._var[c, :] = X_c.var(axis=0)
            self._priors[c, :] = X_c.shape[0] / float(n_samples)

    def predict(self, X):
        y_pred = [self._predict(x) for x in X]
        return(y_pred)
```

```
ef - predict (self, x):
    posteriors = []
    for idx, c in enumerate (self.-classes):
        prior = np.log (self.-priors [idx])
        class-conditional = np.sum (np.log (self.-pdf (idx, x)))
        posterior = prior + class-conditional
        posteriors.append (posterior)
    return (self.-classes [np.argmax (posteriors)])

def -pdf (self, class-idx, x):
    mean = self.-mean [class-idx]
    var = self.-var [class-idx]
    numerator = np.exp (-(x-mean)**2 / (2*var))
    denominator = np.sqrt (2*np.pi*var)
    return (numerator / denominator)
```

PCA Algorithm

```
class PCA:
    def __init__(self, n_components):
        self.n_components = n_components
        self.components = None
        self.mean = None

def fit(self, X):
    self.mean = np.mean(X, axis = 0)
    X = X - self.mean

    cov = np.cov(X.T)

    eigenvalues, eigenvectors = np.linalg.eig(cov)
    eigenvectors = eigenvectors.T
    idxs = np.argsort(eigenvalues)[::-1]

    eigenvalues = eigenvalues[idxs]
    eigenvectors = eigenvectors[idxs]

    self.components = eigenvectors[0:self.n_components]  # store first n components

def transform(self, X):
    # project data
    X = X - self.mean
    return (np.dot(X, self.components.T))
```

~ Neural Network

```
rt numpy as np
n sklearn.metrics import accuracy_score.

def relu(z):
    return np.maximum(z, 0)

ef relu_prime(z):
    return (z>0).astype(z.dtype)

ef sigmoid(z):
    return 1/(1+np.power(np.e, -z))

ef sigmoid_prime(z):
    return Z*(1-Z)

def leaky_relu(z, alpha=0.01):
    return np.where(z>0, z, z*alpha)

def leaky_relu_prime(z, alpha=0.01):
    dz = np.ones_like(z)
    dz[z<0] = alpha
    return dz.

def tanh(z):
    return np.tanh(z)

def tanh_prime(z):
    return 1-(tanh(z)**2).
```

```
def get_activation_function(name):
    if name == 'relu':
        return relu
    elif name == 'sigmoid':
        return sigmoid
    elif name == 'leaky-relu':
        return leaky-relu
    elif name == 'tanh':
        return tanh
    else:
        raise ValueError('Only "relu", "leaky-relu", "tanh" and "
                                                    supp
```

```
def get_derivative_activation_function(name):
    if name == 'relu':
        return relu_prime
    elif name == 'sigmoid':
        return sigmoid_prime
    elif name == 'leaky-relu':
        return leaky_relu_prime
    elif name == 'tanh':
        return tanh_prime
    else:
        raise ValueError("Only "relu", "leaky-relu", "tanh" and "sigmoid
                                                    supported')
```

```
initialize-nn(X, y, hidden-nodes = 4, random-state = None):

    if random-state != None:
        np.random.seed(random-state)

    n-x = X.shape[1]
    n-h = hidden-nodes
    n-y = y.shape[1]

    W1 = np.random.randn(n-x, n-h) # np.sqrt(1./n-h)
    b1 = np.zeros((1, n-h))
    W2 = np.random.randn(n-h, n-y) # np.sqrt(1./n-y)
    b2 = np.zeros((1, n-y))

    params = {
        'W1': W1,
        'b1': b1,
        'W2': W2,
        'b2': b2
    }

    return params
```

```
def forward_prop(X, nn_params, activation = 'relu'):

    W1 = nn_params['W1']
    b1 = nn_params['b1']
    W2 = nn_params['W2']
    b2 = nn_params['b2']

    Z1 = np.dot(X, W1) + b1
    A1 = get_activation_function(activation)(Z1)
    Z2 = np.dot(A1, W2) + b2
    A2 = get_activation_function('sigmoid')(Z2)

    result = {
        'Z1': Z1,
        'A1': A1,
        'Z2': Z2,
        'A2': A2
    }

    return result
```

```
calculate_loss(A2, y):

    m = y.shape[0]
    return np.squeeze(-(1./m) * np.sum(np.multiply(y, np.log(A2)) +
                                       np.multiply(np.log(1-A2), 1-y)))
                # squeeze will convert [[cost]] to 'cost'
                                                  float variable

def backward_prop(X, y, nn_params, cache, learning_rate = 0.001,
                                                  activation = 'relu'):

    W1 = nn_params['W1']
    b1 = nn_params['b1']
    W2 = nn_params['W2']
    b2 = nn_params['b2']

    Z1 = cache['Z1']
    A1 = cache['A1']
    Z2 = cache['Z2']
    A2 = cache['A2']

    m = X.shape[0]

    dz2 = A2 - y
    dw2 = (1./m) * np.dot(A1.T, dz2)
    db2 = (1./m) * np.sum(dz2, axis=0, keepdims = True)
```

```
dz1 = np.dot(dz2, W2.T) * get_derivative_activation_function
                                    (activation)(A1)

dw1 = (1./m) * np.dot(X.T, dz1)
db1 = (1./m) * np.sum(dz1, axis = 0, keepdims = True)

updates = {
    'dw1': dw1,
    'db1': db1,
    'dw2': dw2,
    'db2': db2
    }
    return updates

def update_weights(nn_params, updates, learning_rate = 0.01):
    W1 = nn_params['W1']
    b1 = nn_params['b1']
    W2 = nn_params['W2']
    b2 = nn_params['b2']

    dw1 = updates['dw1']
    db1 = updates['db1']
    dw2 = updates['dw2']
    db2 = updates['db2']
```

```
W1 = W1 - learning-rate * dW1

b1 = b1 - learning-rate * db1

W2 = W2 - learning-rate * dW2

b2 = b2 - learning-rate * db2

final-result = {

    'W1': W1,

    'b1': b1,

    'W2': W2,

    'b2': b2

    }

    return final-result.
```

```
def predict(parameters, X, y):

    cache = forward-prop(X, parameters, 'relu')

    predictions = cache['A2'] > 0.5

    return predictions.
```

```
= 10000
arning_rate = 0.01
hidden_nodes = 50
nn_params = initialize_nn(X, y, hidden_nodes = hidden_nodes, random_state = 0)

history = []
or i in range(epochs+1):
    result = forward_prop(X_train, nn_params, 'relu')
    loss = calculate_loss(result['A2'], y_train)
    history.append(loss)
    updates = backward_prop(X_train, y_train, nn_params, result, learning_rate=0.
                                                        activation = 'relu')
    nn_params = update_weights(nn_params, updates, learning_rate)
if i%(epochs/10) == 0:
    print('Epoch: {} \t Loss: {:.6f} \t Train Accuracy: {:.3f}\t Test
                                        Accuracy: {:.3f}'.format(i, loss,
    accuracy_score(y_train, predict(nn_params, X_train, y_train)),
    accuracy_score(y_test, predict(nn_params, X_test, y_test)))
```

K Means

Outline:

Assume we have input data points $x_1, x_2, x_3, \ldots, x_n$ and the v
of K (the number of clusters). We follow the procedure:

1. Pick K points as the initial centroids from the dataset, either randomly or the first K.
2. find the euclidean distance of each point in the dataset with t identified K points (cluster centroids).
3. Assign each data point to the closest centroid using the distance calculated in the previous step.
4. Find the new centroid by taking the average of the points in each cluster group.
5. Repeat 2 to 4 for a fixed number of iterations or till the centroids don't change.

```
ears :
    def __init__(self, k=3, max_iterations = 500):
        self.k = k
        self.max_iterations = max_iterations

def euclidean_distance(self, point1, point2):
    #return math.sqrt((point1[0] - point2[0])**2 + (point1[1] - point2[1])**2)
    return np.linalg.norm(point1 - point2, axis = 0)

def fit(self, data):
    # let the first K points from the dataset be the initial centroids
    self.centroids = {}
    for i in range(self.k):
        self.centroids[i] = data[i]

# start k-mean clustering.
    for i in range(self.max_iterations):
        # create classes the size of k.
        self.classes = {}
        for j in range(self.k):
            self.classes[j] = []

# find the distance between the points and the centroids
        for point in data:
            distances = []
            for index in self.centroids.
                distances.append(self.euclidean_distance(point, self.centroids[index]))
```

```
# start k-means clustering.
for i in range (self.max_iterations):
    # create classes the size of k
    self.classes = {}

    for j in range (self.k):
        self.classes [j] = [].

    # find the distance between the points and the centroids.
    for point in data:
        distances = []
        for index in self.centroids:
            distances.append (self.euclidean_distance (point,
                                                       self.centroids [index]))
        # find which cluster the datapoint belongs to by finding the
        # min distance.
        cluster_index = distances.index (min (distances))
        self.classes [cluster_index].append (point)

    # now that we have classified the datapoints into clusters,
    # we need to again find the new centroid by taking the
    # average of the points in the cluster class.
    for cluster_index in self.classes:
        self.centroids [cluster_index] = np.average (self.classes [cluster_
                                                                   index],
                                                     axis = 0)
```

ent Descent

E cost function for linear regression model

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

m training examples: $\left\{ (x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)}) \right\}$.

$x^{(i)} \in \mathbb{R}^{n_x} \ \forall \ i$

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}_{n_x}$$

$\longleftarrow m \longrightarrow$

$X \in \mathbb{R}^{n_x \times m}$.

$$y = \begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix} \in \mathbb{R}^{1 \times m}.$$

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_{n_x} \end{bmatrix}$$

To find the value of θ that minimizes the cost function, there is a closed form solution.

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

Batch Gradient Descent

Partial derivatives of the cost function.

$$\frac{\partial}{\partial \theta_j} MSE(\theta) = \frac{2}{m} \sum_{i=1}^{m} \left(\theta^T x^{(i)} - y^{(i)}\right) x_j^{(i)}$$

Gradient vector of the cost function

$$\nabla_\theta MSE(\theta) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} MSE(\theta) \\ \frac{\partial}{\partial \theta_1} MSE(\theta) \\ \frac{\partial}{\partial \theta_n} MSE(\theta) \end{pmatrix}$$

$$= \frac{2}{m} X^T (X \cdot \theta - y)$$

Gradient Descent step

$$\theta^{(\text{next step})} = \theta - \eta \nabla_\theta MSE(\theta).$$

```
eta = 0.1  # learning rate.
n_iterations = 1000
m = 100
theta = np.random.randn(2,1).
for iteration in range(n_iterations):
    gradients = 2/m * X_b.T.dot(X_b.dot(theta) - y)
    theta = theta - eta * gradients.
```

ric Gradient Descent.

epochs = 50.

0, t1 = 5, 50 # learning schedule hyperparameters.

ef learning-schedule (t):

return t0/(t+t1)

theta = np.random.randn(2,1) # random initialization

for epoch in range(n-epochs):

for i in range(m):

random-index = np.random.randint(m)

xi = X-b[random-index : random-index+1]

yi = y[random-index : random-index+1]

gradients = 2 * xi.T.dot(xi.dot(theta) - yi)

eta = learning-schedule(epoch*m+i)

theta = theta - eta * gradients.

Mini-Batch Gradient Descent

```
def get_batches (X, y, batch_size, i):
    X_new = X [i : i+batch_size, :]
    y_new = y [i : i+batch_size]
    return X_new, y_new .

def MiniBatchGradientDescent (X, y, theta, learning_rate = 0.1, batch_size = 10):
    num_batches = int ( X.shape[0] / batch_size )

    gradient = 0

    for i in range (0, num_batches):
        X_batch, y_batch = get_batches (X, y, batch_size, i)
        hypothesis = np.dot ( X_batch, theta)
        loss = hypothesis - y_batch
        X_trans = X_batch.T
        gradient = np.dot (X_trans, loss) / batch_size
        theta = theta - learning_rate * gradient

    return theta .
```