Adaline Classifier with Stochastic Gradient Descent: Python Implementation and Insights

- **Introduction**
  - AdalineSGD:
    - An adaptive linear neuron model, learning weights via stochastic gradient descent (SGD)—updates weights after each training example.
  - Stochastic Gradient Descent (SGD):
    - Faster and more scalable than batch gradient descent, especially effective for large datasets or online learning.

- **Theory: How AdalineSGD Works**
  - SGD vs. Batch GD:
    - Batch GD: Updates after full dataset (slow for large data).
    - SGD: Updates after each training example—much faster, especially on big data/streaming data.
  - Cost Function:
    - Same as AdalineGD (Sum of Squared Errors), but average cost tracked over mini batches/epochs.
  - Randomization (Shuffling):
    - Shuffles data before each epoch to avoid cycles and improve convergence.
  - Feature Scaling:
    - Standardization (mean=0, std=1) is essential for stable learning.

- **Key Equations and Diagrams**
  - Weight Update (SGD):

$$w := w + \eta(y^{(i)} - \phi(z^{(i)}))x^{(i)}$$

for each training example i.

  - Mini-batch SGD:
    - Uses small batches (e.g., 32, 64 samples) to balance speed and stability.

- **Step-by-Step Python Implementation**

Adaline Classifier with Stochastic Gradient Descent: Python Implementation and Insights

```python
# Adaline with Stochastic Gradient Descent on the Iris Dataset

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# 1. Load the dataset
```

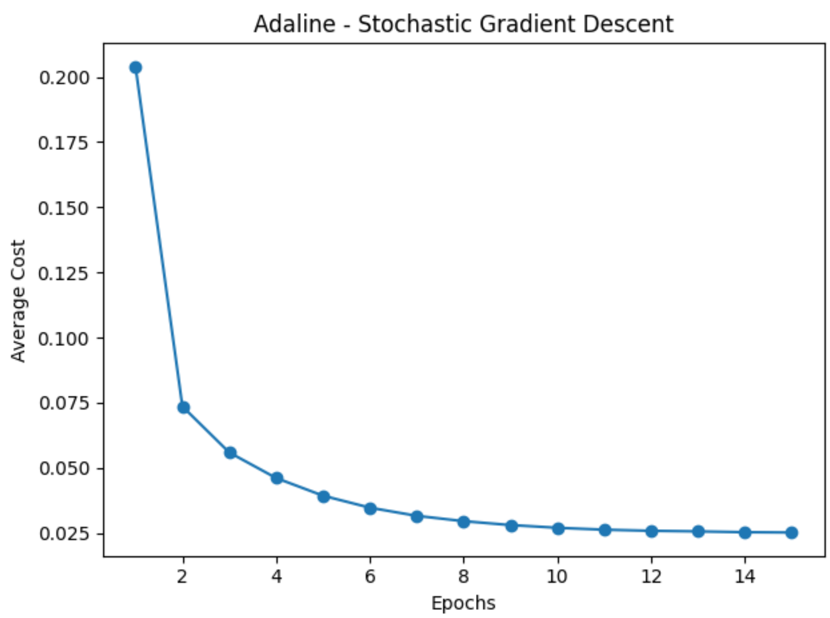
```python
df = pd.read_csv(url, header=None)
y = df.iloc[0:100, 4].values
y = np.where(y == 'Iris-setosa', -1, 1)
X = df.iloc[0:100, [0, 2]].values

# 2. Standardize features
X_std = np.copy(X)
X_std[:, 0] = (X[:, 0] - X[:, 0].mean()) / X[:, 0].std()
X_std[:, 1] = (X[:, 1] - X[:, 1].mean()) / X[:, 1].std()

# 3. AdalineSGD implementation
class AdalineSGD(object):  1 usage
    """ADAptive LInear NEuron classifier with Stochastic Gradient Descent."""

    def __init__(self, eta=0.01, n_iter=15, shuffle=True, random_state=None):
        self.eta = eta
        self.n_iter = n_iter
        self.shuffle = shuffle
        self.random_state = random_state

    def fit(self, X, y):  1 usage
        self._initialize_weights(X.shape[1])
        self.cost_ = []
        for i in range(self.n_iter):
            if self.shuffle:
                X, y = self._shuffle(X, y)
            cost = []
            for xi, target in zip(X, y):
                cost.append(self._update_weights(xi, target))
            avg_cost = sum(cost) / len(y)
            self.cost_.append(avg_cost)
        return self

    def partial_fit(self, X, y):
        if not hasattr(self, 'w_initialized'):
            self._initialize_weights(X.shape[1])
        if y.ravel().shape[0] > 1:
            for xi, target in zip(X, y):
                self._update_weights(xi, target)
        else:
            self._update_weights(X, y)
        return self
```

Adaline Classifier with Stochastic Gradient Descent: Python Implementation and Insights

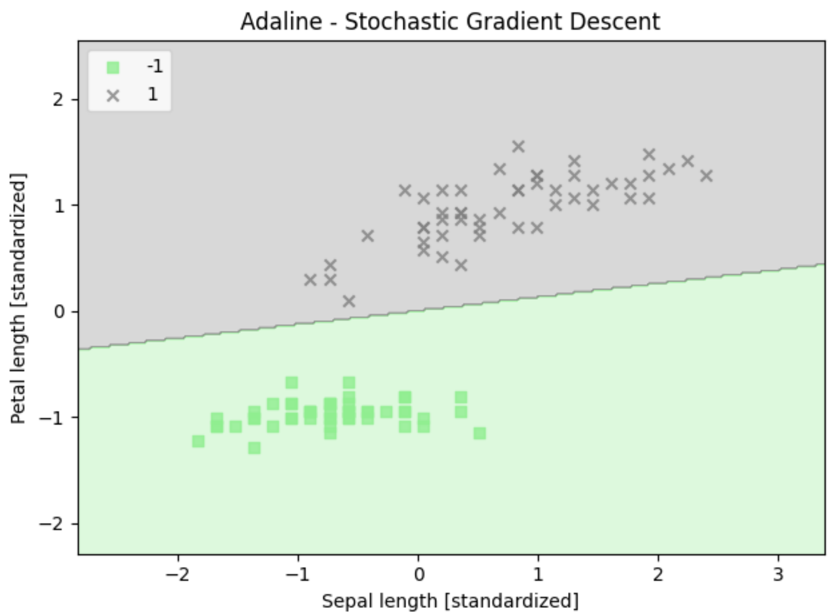
```
    def _shuffle(self, X, y):  1 usage
        r = self.rgen.permutation(len(y))
        return X[r], y[r]

    def _initialize_weights(self, m):  2 usages
        self.rgen = np.random.RandomState(self.random_state)
        self.w_ = self.rgen.normal(loc=0.0, scale=0.01, size=1 + m)
        self.w_initialized = True

    def _update_weights(self, xi, target):  3 usages
        output = self.activation(self.net_input(xi))
        error = target - output
        self.w_[1:] += self.eta * xi * error
        self.w_[0] += self.eta * error
        cost = 0.5 * error**2
        return cost

    def net_input(self, X):  2 usages
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, X):  2 usages
        """Identity activation (linear)"""
        return X

    def predict(self, X):  3 usages (3 dynamic)
        """Return class label after unit step"""
        return np.where(self.activation(self.net_input(X)) >= 0.0, 1, -1)

# 4. Train the model
ada_sgd = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
ada_sgd.fit(X_std, y)

# 5. Plot average cost over epochs
plt.plot( *args: range(1, len(ada_sgd.cost_) + 1), ada_sgd.cost_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Average Cost')
plt.title('Adaline - Stochastic Gradient Descent')
plt.tight_layout()
plt.show()

# 6. Plot decision regions
from matplotlib.colors import ListedColormap
```

Adaline Classifier with Stochastic Gradient Descent: Python Implementation and Insights

```python
from matplotlib.colors import ListedColormap

def plot_decision_regions(X, y, classifier, resolution=0.02):  1 usage
    markers = ('s', 'x')
    colors = ('lightgreen', 'gray')
    cmap = ListedColormap(colors[:len(np.unique(y))])
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf( *args: xx1, xx2, Z, alpha=0.3, cmap=cmap)
    plt.xlim( *args: xx1.min(), xx1.max())
    plt.ylim( *args: xx2.min(), xx2.max())
    # plot class samples
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1],
                    alpha=0.8, c=colors[idx],
                    marker=markers[idx], label=cl)

plot_decision_regions(X_std, y, classifier=ada_sgd)
plt.title('Adaline - Stochastic Gradient Descent')
plt.xlabel('Sepal length [standardized]')
plt.ylabel('Petal length [standardized]')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```



Adaline Classifier with Stochastic Gradient Descent: Python Implementation and Insights



❖ **Results & Discussion**

- Plots: Average cost vs. epochs; decision boundary.
- Discussion:
  - Compare convergence speed and plot smoothness to AdalineGD.
    - AdalineSGD converged faster in terms of wall-clock time, since weights were updated after every sample. However, the average cost per epoch plot was noisier compared to the smooth decrease in AdalineGD, due to the randomness of sample ordering and weight updates.
  - Note impact of shuffling and online updates.
    - Shuffling the training data before each epoch helped prevent learning cycles and improved overall convergence stability. Online updates (partial_fit) enable the model to learn from new data in real-time, which is essential for streaming or continuously updated datasets.
  - Discuss real-world use for large-scale/streaming data.
    - Stochastic Gradient Descent is well-suited for large-scale and streaming applications, where it’s impractical to load all data at once. The ability to update weights incrementally means AdalineSGD can adapt to new data efficiently, which is a key requirement in real-world systems such as recommendation engines, financial modeling, and IoT analytics.

Adaline Classifier with Stochastic Gradient Descent: Python Implementation and Insights

- **Connecting to Advanced ML**
  - SGD in Deep Learning:
    - Foundation for training neural nets and large models; modern optimizers (Adam, RMSProp) are based on SGD.
  - Online & Mini-batch Learning:
    - Mini-batch SGD is industry standard for deep learning.
  - When to use SGD:
    - Essential for large, streaming, or continually updated datasets.

- **References**

    - Python Machine Learning by Sebastian Raschka
    - scikit-learn documentation