# Regularization and Overfitting

## Jack Bressett

### January 25, 2020

## Why Regularize?

**In its essence, regularizing is really just adding more error into your model. The difficult part is finding the perfect balance of hyper parameters that guide that error into effective deployment.**

**Regularizing can take many forms depending on the type of learning task at hand, but the goal remains constant: reduce overfitting. When faced with a model that is performing too well on the training data and very poorly on the validation set, a data scientist must be able to resolve the gap. One way to achieve this is by adding more error into the training data so that the model is more prepared to take on generalized data. This technique is regularization and has proven effective for many learners, especially within regression tasks.**

### We are going to begin with our essential imports:

```
In [1]: import matplotlib.pyplot as plt
        import numpy as np
        from sklearn.preprocessing import StandardScaler
        from sklearn.linear_model import LinearRegression
        from sklearn.linear_model import SGDRegressor
        from sklearn.pipeline import Pipeline
        from sklearn.preprocessing import PolynomialFeatures
        from sklearn.linear_model import Ridge
        from sklearn.linear_model import Lasso
        from sklearn.linear_model import ElasticNet
```

### We next want a function to help visualize our regulizers. Fortunately our book provides us with one:

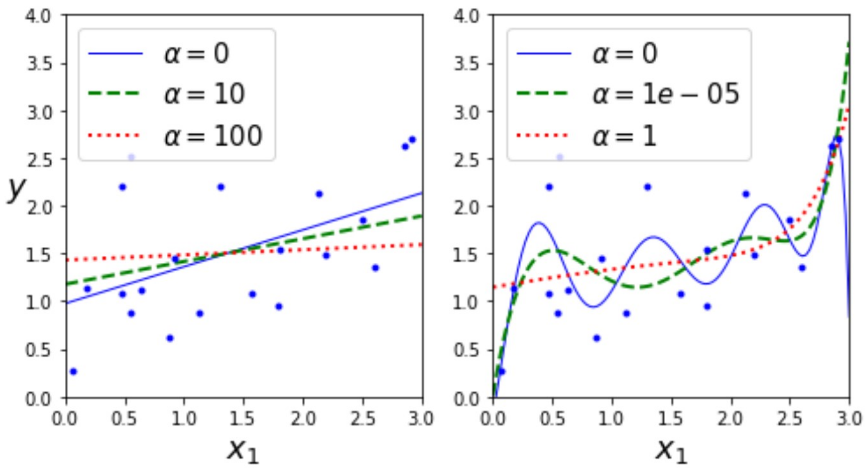
```
In [2]: def plot_model(model_class, polynomial, alphas, **model_kargs):
            for alpha, style in zip(alphas, ("b-", "g--", "r:")):
                model = model_class(alpha, **model_kargs) if alpha > 0 else LinearRegressio
        n()
                if polynomial:
                    model = Pipeline([
                            ("poly_features", PolynomialFeatures(degree=10, include_bias=Fa
        lse)),
                            ("std_scaler", StandardScaler()),
                            ("regul_reg", model),
                        ])
                model.fit(X, y)
                y_new_regul = model.predict(X_new)
                lw = 2 if alpha > 0 else 1
                plt.plot(X_new, y_new_regul, style, linewidth=lw, label=r"$\alpha = {}$".fo
        rmat(alpha))
            plt.plot(X, y, "b.", linewidth=3)
            plt.legend(loc="upper left", fontsize=15)
            plt.xlabel("$x_1$", fontsize=18)
            plt.axis([0, 3, 0, 4])
```

## Now let's generate some data and a comparison plot for Ridge Regularization

```
In [3]: np.random.seed(42)
        m = 20
        X = 3 * np.random.rand(m, 1)
        y = 1 + 0.5 * X + np.random.randn(m, 1) / 1.5
        #Refit our data to our plot dimensions
        X_new = np.linspace(0, 3, 100).reshape(100, 1)

        plt.figure(figsize=(8,4))
        plt.subplot(121)
        plot_model(Ridge, polynomial=False, alphas=(0, 10, 100), random_state=42)
        plt.ylabel("$y$", rotation=0, fontsize=18)
        plt.subplot(122)
        plot_model(Ridge, polynomial=True, alphas=(0, 10**-5, 1), random_state=42)

        plt.show()
```

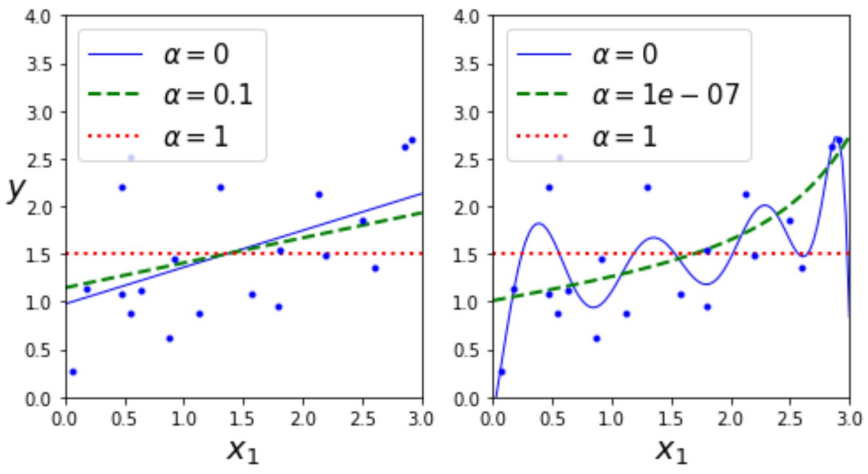


**On the left we have a linear model fit with Ridge's method and on the right is a tenth degree polynomial with the same data. Note the hyper parameter alpha that is varying discretely over (0,10,100) for the linear model and (0,1e-05,1) for the polynomial model. Note that for both plots the models with the largest alpha values are the most stable and tend toward the arithmetic mean of the data. This is a product of the mathematical architecture at play.**

## Next we want to take a look at the Lasso method:

```
In [4]: plt.figure(figsize=(8,4))
        plt.subplot(121)
        plot_model(Lasso, polynomial=False, alphas=(0, 0.1, 1), random_state=42)
        plt.ylabel("$y$", rotation=0, fontsize=18)
        plt.subplot(122)
        plot_model(Lasso, polynomial=True, alphas=(0, 10**-7, 1), tol=1, random_state=42)

        plt.show()
```



**Here we see that the Lasso method is adding much more error than the Ridge method. This is evident in the change of scale in the linear alpha domain and in the amount of change generated from a very small step in the polynomial model. After that tiny step the curve has almost completely flattened out.**

**These methods represent seemingly opposite sides of the spectrum. One seems to add small amounts of error at a time and the other seems to be a bit more liberal. Fortuntaly for perfectionists there is another option available. By utilizing an additional hyper parameter, Elastic Net adds error as a combination of the two methods together to ensure an error hybrid crafted to fit inbetween both regulizers. This can be seen numerically with the following code:**

In [9]:
```
#Ridge Predictor
ridge_reg = Ridge(alpha=1, solver="sag", random_state=42)
ridge_reg.fit(X, y)
print("Ridge Output: ", ridge_reg.predict([[1.5]]))

#Lasso Predictor
lasso_reg = Lasso(alpha = .1)
lasso_reg.fit(X,y)
print("Lasso Output: ",lasso_reg.predict([[1.5]]))

#Elastic Net Predictor
elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5, random_state=42)
elastic_net.fit(X, y)
print("Elastic Net Output: ",elastic_net.predict([[1.5]]))
```

```
Ridge Output:  [[1.5507201]]
Lasso Output:  [1.53788174]
Elastic Net Output:  [1.54333232]
```

**All of these values in our case are good because they all alleviate the problem of overfitting. The next step is to identify with of these methods you want to adopt into your model. In other words, how much error is best for you?**