Module 3: Supervised Learning Problems

3-1 Gradient Descent:

When discussing model performance, we often cite a **Evaluation Metric** such as an **MSE** or R^2 score, of which is minimized or maximized for best performance. Most models have a set of **Parameters/Weights** they tweak to tune the performance on this **Evaluation Metric**, but; *How do they decide how to tweak these parameters?* To do so amounts to finding an algorithm which can find the minimum/maximum of an arbitrary function $f(X_1, X_2, X_3, \cdots, X_N)$.

In order to do this, we borrow a concept from Multivariable Calculus: the **Gradient** ∇, which is defined as the vector pointing in the direction of the fastest rate of increase of the function at a given point such that:

$$P_t = \langle x_1, x_2, x_3, \cdots, x_n \rangle_t$$

$$.$$

$$P_{t+1} = P_t - \eta \overrightarrow{\nabla} f(P_t)$$

$$.$$

$$Pr(f(P_{t+1}) < f(P_t)) \approx 1$$

$$.$$

$$f(P_{t=\infty}) \to min(f(X_1, X_2, X_3, \cdots, X_N))$$

In other words, if we continue to calculate the **Gradient** and use that to update the function parameters iteratively, we should converge onto the minimum of the function. Note that η is a **Hyperparameter** which represents the **Learning Rate**, which scales magnitude of the steps we take and is typically should relate to our confidence in the step. This process of minimizing a function is known as **Gradient Descent**

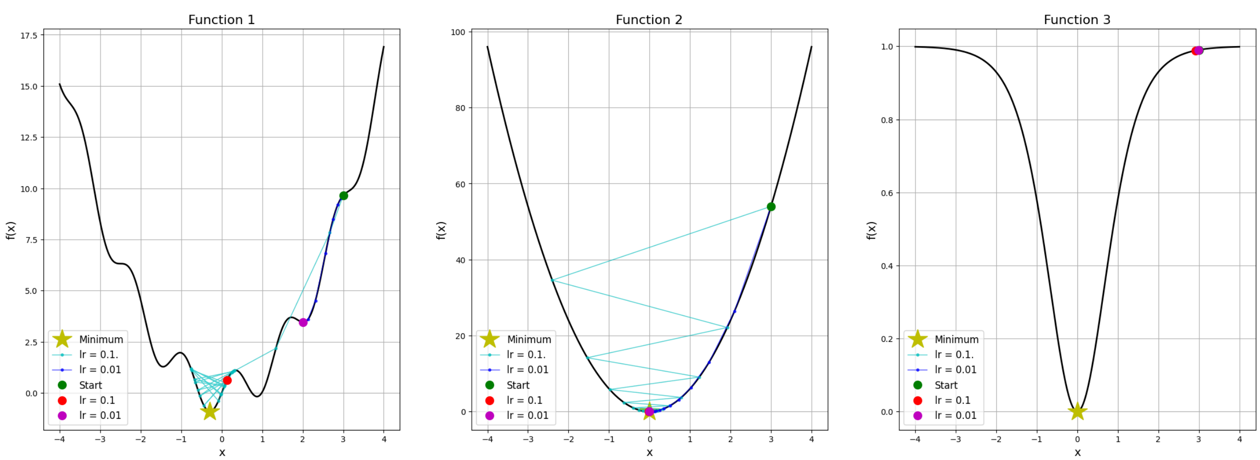
Note that this Constant **Learning Rate** or Plain **Gradient Descent** is unlikely to converge on the true **Global Minimum** if there are multiple **Local Minima** or very shallow regions containing minima as it gets 'stuck'.

Example: Code a **Class** that implements the basic **Gradient Descent** Algorithm on three different functions at both a high and low learning rate. Plot the resulting paths.

```python
import numpy as np
import matplotlib.pyplot as plt

class BasicGradD:
    def __init__(self):
        pass

    def __grad(self, f, x, h=1e-8):
        #Compute gradient of f at x
        dfdx = (f(x + h) - f(x - h))/(2*h)
        return dfdx

    def gradient_descent(self, f, x0, eta=0.01, tol=1e-6, max_iter=25):
        '''Basic Gradient Descent Algorithm for a single variable function y = f(x)'''
        #Initialization
        self.path = [x0]
        x = x0

        #Update Loop
        for n in range(max_iter):
            x_new = x - eta*self.__grad(f, x)
            self.path.append(x_new)
            #Check for Convergence
            if abs(x_new - x) < tol:
                break
            x = x_new

        return self.path

    def plot(self, *args):
        '''Plot the functions and the path taken by gradient descent'''
        #Lrs
        lr_high = 0.1
        lr_low = 0.01

        #Formatting params
        se_size = 10

        #Define Paths
        funcs = args[0::3]
        paths_highlr = args[1::3]
        paths_lowlr = args[2::3]
        num_paths = len(paths_highlr)

        #Calc mins
        x = np.linspace(-4, 4, 1000)
        mins = x[np.argmin([f(x) for f in funcs], axis=1)]
        mins_vals = [f(mins[i]) for i, f in enumerate(funcs)]

        #Define figure
        fig, ax = plt.subplots(1, num_paths, figsize=(27, 9))

        #Plot each function
        for i, (f, path_high, path_low) in enumerate(zip(funcs, paths_highlr, paths_lowlr)):
            y = f(x)
            ax[i].plot(x, y, 'k-', lw=2) #Function
            ax[i].plot(mins[i], mins_vals[i], 'y*', markersize=25, label='Minimum') #Minimum point
            ax[i].plot(path_high, f(np.array(path_high)), 'co-', lw=1, markersize=3, alpha=0.7, label=f'lr = {lr_high}.') #High LR Path
            ax[i].plot(path_low, f(np.array(path_low)), 'bo-', lw=1, markersize=3, alpha=0.7, label=f'lr = {lr_low}') #Low LR Path
            ax[i].plot(path_high[0], f(path_high[0]), 'go', markersize=se_size, label='Start') #Starting point
            ax[i].plot(path_high[-1], f(path_high[-1]), 'ro', markersize=se_size, label=f'lr = {lr_high}') #High LR Ending point
```

```python
            ax[i].plot(path_low[-1], f(path_low[-1]), 'mo', markersize=se_size, label=f'lr = {lr_low}') #Low LR Ending point
            ax[i].set_title(f'Function {i+1}', fontsize=16)
            ax[i].set_xlabel('x', fontsize=14)
            ax[i].set_ylabel('f(x)', fontsize=14)
            ax[i].grid(True)
            ax[i].legend(fontsize=12)



#Functions to minimize
f1 = lambda x: x**2 + np.sin(5*x)
f2 = lambda x: 6*x**2
f3 = lambda x: np.tanh(x)**2

#Create instance of BasicGradD class
bgd1 = BasicGradD()

#Compute gradient descent paths
start = 3
f1_path_high = bgd1.gradient_descent(f1, x0=start, eta=0.15)
f2_path_high = bgd1.gradient_descent(f2, x0=start, eta=0.15)
f3_path_high = bgd1.gradient_descent(f3, x0=start, eta=0.15)
f1_path_low = bgd1.gradient_descent(f1, x0=start, eta=0.025)
f2_path_low = bgd1.gradient_descent(f2, x0=start, eta=0.025)
f3_path_low = bgd1.gradient_descent(f3, x0=start, eta=0.025)

#Plot the functions and paths
bgd1.plot(f1, f1_path_high, f1_path_low, f2, f2_path_high, f2_path_low, f3, f3_path_high, f3_path_low)
```

Output:



Notice that the **Low Learning Rate** gets caught in $f1$'s **Local Minima**, the **High Learning Rate** overshoots $f2$'s **Global Minima** and oscillates around it, and both **Learning Rates** struggle with the shallow region due to the **Vanishing Gradient**.

In order to prevent the algorithm from being caught in **Local Minima** and to help with any **Vanishing Gradients**, we introduce a small **Stochastic (Random)** component to the algorithm. This can either be done by applying slight random nudges to the evaluation point (in the case of a traditional function) or by calculating the **Gradient** on random subsets of the data (for model optimization), known as **Stochastic Mini-Batch Gradient Descent**. This also results in more steps being taken and should lead to faster convergence.

Additionally, you can implement a **Adaptive Learning Rate** in which the **Learning Rate** changes as training proceeds according to some predetermined function. These are also known as **Schedulers** and some common ones can be seen below:

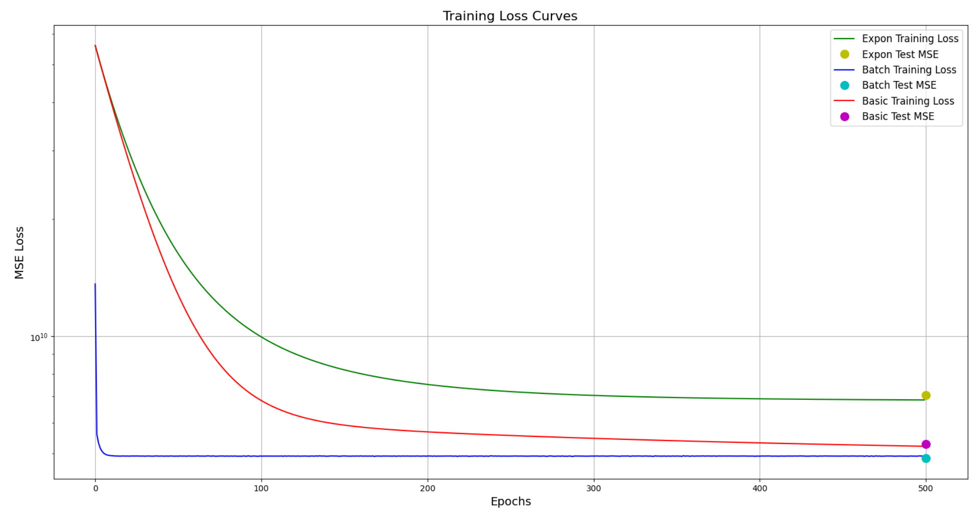
Exponential Decay: $\eta_t = \eta_0 e^{-\lambda t}$

Polynomial Decay: $\eta_t = \eta_0(\beta t + 1)^{-\alpha}$

Example: Code a **Class** which implements **Basic Gradient Descent**, **Stochastic Mini-Batch Gradient Descent**, and an **Exponential Decay Adaptive Learning Rate** to optimize a **Linear Model** on the housing dataset (Generation code in the matching notebook).

Output:

```
Expon
Expon R^2: 0.4819, MSE: 7.059e+09
Batch R^2: 0.6430, MSE: 4.864e+09
Basic R^2: 0.6119, MSE: 5.288e+09
```



Notice that both the **Exponential Decay LR Gradient Descent** and the **Batched Gradient Descent** converged far quicker than the basic implementation. Also, note that the **Testing Loss** is higher than the **Training Loss** suggesting there is some slight **Overfitting**.

3-2 Adaptive Gradients (Optimizers):

While **Adaptive Learning Rates** do help, they are predefined as a function of the epochs, which leads to inefficiencies as it is not accounting for any of the retrieved data on the **Gradient**. Thus, the next step would be to implement a way to adjust the **Learning Rate** based on the calculated **Gradient Information**.

The simplest implementation of this would be to adjust the **Learning Rate** inversely with the magnitude of the **Gradient**, which helps prevent overshooting and oscillations when the **Gradient** is steep. This method is know as the **ADAGrad Optimizer**.

To accomplish this, an accumulation term s is introduced, which gathers the data on the magnitude of the **Gradient**:

ADAGrad:

$$s_t = s_{t-1} + g_t^2$$
.
$$w_t = w_{t-1} - \frac{\eta}{\sqrt{s_{t-1} + \epsilon}} g_{t-1}$$
.
$$\eta_{effective} = \frac{\eta}{\sqrt{s_t + \epsilon}}$$

To improve upon this implementation, which leads to a strictly increasing s as the **Gradient Magnitude** accumulates, we can introduce a sort of Decay term $\gamma \in (0 - 1) \approx 1$ that modulates the previous magnitude terms every step. This, the accumulation term consists of a smoothly decaying series of the previous magnitude terms:

RMSProp:

$$s_t = \gamma s_{t-1} + (1 - \gamma) g_t^2$$
.

$$s_t = (1-\gamma)(g_t^2 + \gamma g_{t-1}^2 + \gamma^2 g_{t-2}^2 + \cdots + \gamma^t g_0^2)$$
.
$$w_t = w_{t-1} - \frac{\eta}{\sqrt{s_{t-1}+\epsilon}} g_{t-1}$$

Finally, we can implement a **Momentum** term which limits the optimizers ability to change direction each step and account for the current 'velocity':

ADAM:

$$v_t = \beta_1 v_{t-1} + (1-\beta_1) g_t$$
.
$$s_t = \beta_2 s_{t-1} + (1-\beta_2) g_t^2$$
.
$$\hat{v}_t = \frac{v_t}{1-\beta_1^{t+1}}, \hat{s}_t = \frac{s_t}{1-\beta_2^{t+1}}$$
.
$$\hat{g}_t = \frac{\eta \hat{v}_t}{\sqrt{\hat{s}_t}+\epsilon}$$
.
$$w_t = w_{t-1} - \hat{g}_t$$

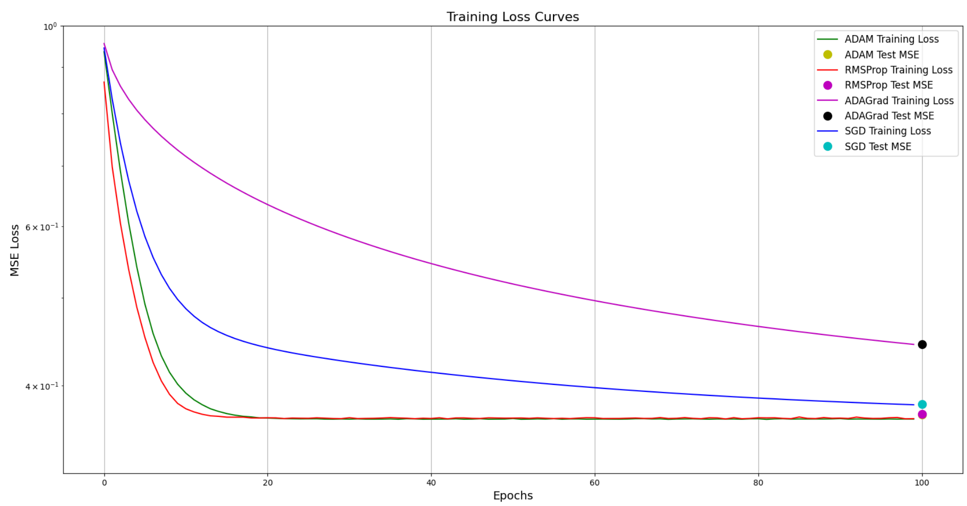
This **Optimizer** is particularly effective on **Sparse Gradients**, which will tend to occur when the number of weights is high, like in **Neural Networks**

Fillin when to use, pros/cons

Example: Add **ADAGrad**, **RMSProp**, and **ADAM** to the previous LinearSMBGD **Class**, use **Mini-Batching** for all. You will likely have to scale the targets as otherwise the magnitude of the **Gradient** explodes (Generation code in the matching notebook).

Output:

```
SGD     R^2: 0.6190, MSE: 0.3817
ADAGrad R^2: 0.5564, MSE: 0.4445
RMSProp R^2: 0.6285, MSE: 0.3722
ADAM    R^2: 0.6294, MSE: 0.3713
```

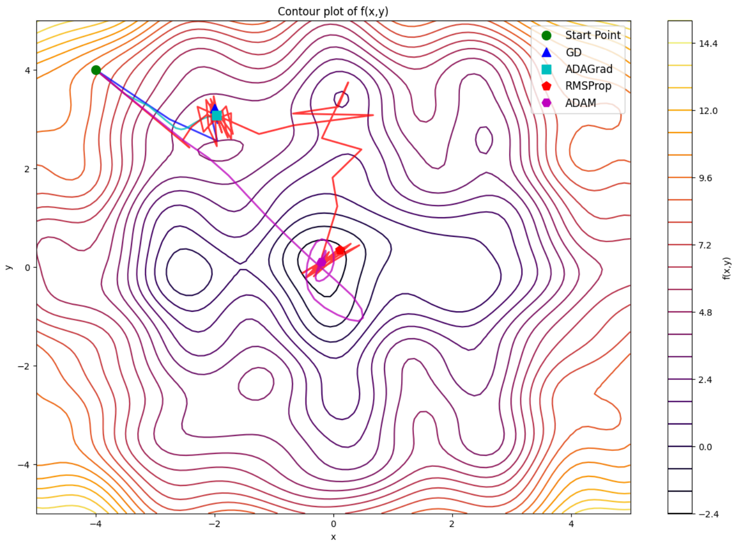


The above is just one example. Depending on the circumstances, the best **Optimizer** will change.

Example: Use the above **Optimizers** to find the minimum of a function $f(x, y)$ (Generation code in the matching notebook).

Output:

```
Optimization Params:  max_iter = 250, eta = 0.5, tol = 1e-08

Start Point:         -4.00, 4.00, f = 9.02

GD End Point:        -2.00, 3.22, f = 3.68
ADAGrad End Point:   -1.97, 3.07, f = 3.64
RMSProp End Point:    0.11, 0.35, f = -1.72
ADAM End Point:      -0.20, 0.10, f = -2.17
```



Notice that **Basic Gradient Descent** and **ADAGrad** get trapped by a local minima whereas **RMSProp** and **ADAM** converge on the global minimum, with **ADAM** taking a much smoother path than **RMSProp**.

3-3 Complexity, Errors, Bias, & Variance:

In machine learning, the relation between **Model Complexity**, **Errors**, **Bias**, and **Variance** is fundamental to evaluating the performance of a model and determining its **Optimality** and ability to **Generalize**.

Bias:
Bias refers to the error introduced by approximating a real-world problem, which may be complex, by a simplified model. **High Bias** can cause an algorithm to miss relevant relations between features and target outputs (**Underfitting**). In other words, the model does not contain the required **Complexity** to model the underlying relations in your dataset.

- **High Bias**: Assumptions in the model are too strong, making it overly simplistic.
- **Low Bias**: The model captures the true relationship more accurately.

Variance:
Variance refers to the error introduced by the model's sensitivity to the fluctuations in the training data. **High Variance** can cause an algorithm to model the random noise in the training data, rather than the intended outputs (**Overfitting**). In other words, the model has been provided too many free parameters and begins to use them to "learn" noise present in the training data.

- **High Variance**: The model is too complex, capturing noise along with the underlying pattern.
- **Low Variance**: The model's predictions are stable across different training sets.

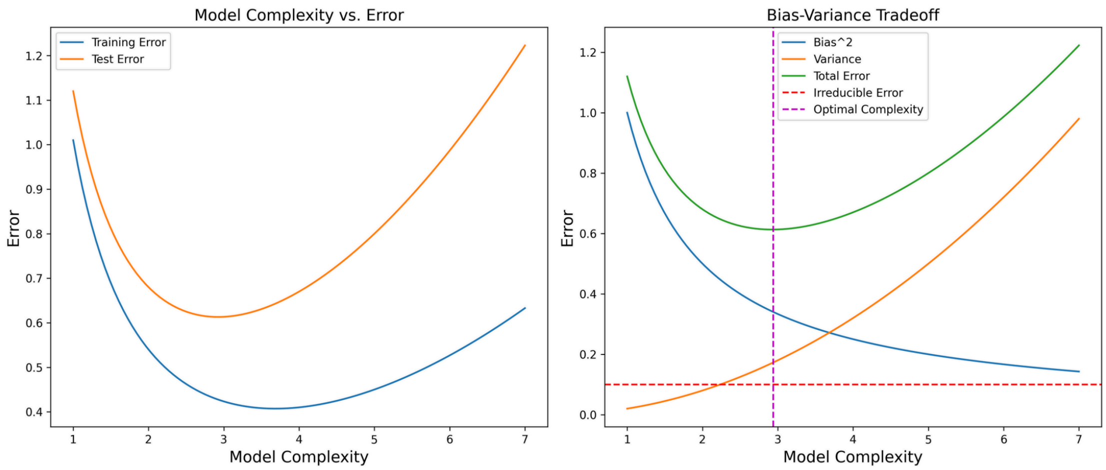
Since both **Bias** and **Variance** relate to model **Complexity**, there exists a tradeoff between the two in which an increase in the **Bias** results in a decrease in the **Variance** and vice-versa.

The **Total Error** of a model can be expressed as the sum of **Bias** squared,
Variance, and **Irreducible Error (Noise)**:

$$\text{Total Error} = (\text{Bias})^2 + \text{Variance} + \text{Irreducible Error}$$

The **Irreducible Error** is due to **Noise** in the data itself and cannot be
reduced by any model. Thus, a model whose **Total Error** matches this term can be considered **Maximally Optimal**, unable to be improved.

Below is a visualization of the **Bias-Variance Tradeoff** as well as its relation to the model's performance on training vs testing data:



The above figure can be interpreted as illustrating the following:

Model Complexity vs. Error:

- The **Training Error** decreases as model **Complexity** increases because the model can better fit the training data with additional free parameters.
- The **Test Error** decreases initially but starts to increase after a certain point, indicating **Overfitting**. The test error curve shows a local minimum, reflecting the optimal model **Complexity** where the tradeoff between **Bias** and **Variance** is balanced.

Bias-Variance Decomposition:

- **Bias** squared decreases with increasing model **Complexity**, as the model becomes more capable of capturing the underlying patterns with additional free parameters.
- **Variance** increases with model **Complexity**, as the model starts to fit the **Noise** present in the training data.
- The **Total Error** curve shows a local minimum at an optimal model **Complexity**, balancing **Bias** and **Variance**. This minimum is marked with a vertical green dashed line.
- The intersection point of the **Bias** and **Variance** curves is not necessarily the point of minimum **Total Error**.

3-4 Regularization & Overfitting:

Linear Models have a prediction equation which takes the following form:

$$y_{pred} = \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n$$

These weights β of these models are determined by minimizing the **Sum of Squared Errors (SSE)**:

$$SSE = \sum_{i=1}^{n} (y - y_{pred})^2$$

Note that we typically use **MSE** in practical implementations so it can be evaluated easily between implementations as it normalizes for the number of samples.

This model framework is know as the **Ordinary Least Squares** method.

This method breaks down if the model is provided with too many **Free Parameters** when compared to the number of samples present in the training data. As an example, if a model were to be provided with 1000 samples and 1000 free parameters, it could conceivably memorize the **Training Targets**.

This is an example of **Overfitting**, denoted by the model memorizing the specific form of the **Training Data** rather than its underlying relationships. This will decrease the model's ability to **Generalize**, or perform accurate predictions on data it has not been trained on (**Testing Data**).

In general for such linear models, you can define a metric for the model **Complexity** as follows:

$$\text{Complexity} \propto \sum_{i=1}^{n} |\beta_i^K|$$

Where **Complexity** relates to the total magnitude of the **Weights**. Note that K is an arbitrary power and is typically limited to $1, 2$ of a combination of both.

In order to reign in model **Complexity** and stave off **Overfitting**, we introduce this metric into our **Loss Function**. This "technique" is know as **Regularization** .

L1 Regularization (Lasso Regression):

In **L1 Regularization**, we introduce the above complexity term with an applied weighting α and $K = 1$:

$$\text{Loss} \propto \text{SSE} + \text{L1 Complexity} = \text{SSE} + \alpha_{L1} \sum_{i=1}^{n} |\beta_i|$$

- This type of **Regularization** will tend to reduce the weights of unimportant **Features** to **Exactly 0**, which has the effect of removing them from the prediction equation.
- This is incredibly useful for **Feature Selection** when you believe there are irrelevant or weakly predictive **Features** present, and allows you to predict the **Target** with less required **Features**.

L2 Regularization (Ridge Regression):

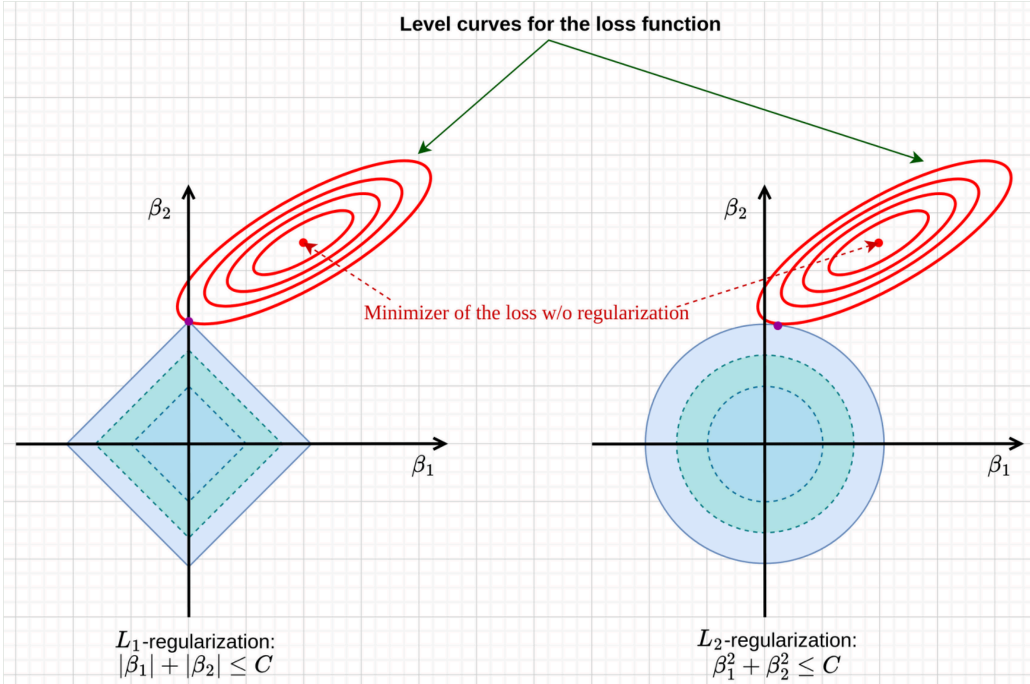
In **L2 Regularization**, we introduce the above complexity term with an applied weighting α and $K = 2$:

$$\text{Loss} \propto \text{SSE} + \text{L2 Complexity} = \text{SSE} + \alpha_{L2} \sum_{i=1}^{n} \beta_i^2$$

- This type of **Regularization** will tend to reduce the weights of unimportant **Features** to **Near 0**. This does not reduce the number of **Features** used, but does reduce model **Complexity**.
- This is useful when you have a large number of **Features** that you believe to all be moderately predictive.

For both of the above **Regularization** types, the parameter α is a measure of the strength of the imposed regularization.

See a visualization of the differences between **L1** and **L2 Regularization** below:



These two types of **Regularization** can also be combined (**Elastic Net Regularization**)

Elastic Net Regularization

In **Elastic Net Regularization** we introduce both an **L1** and **L2 Regularization** term into our **Loss Function**:

$$\text{Loss} \propto \text{SSE} + \alpha\left(\lambda\sum_{i=1}^{n}|\beta_i| + (1-\lambda)\sum_{i=1}^{n}\beta_i^2\right)$$

- This type of **Regularization** can assist in both general **Complexity** reduction as well as **Feature Selection**.

The parameter λ determined the relative strengths of the **L1** and **L2** regularization components, and can be any value between $(0-1)$.

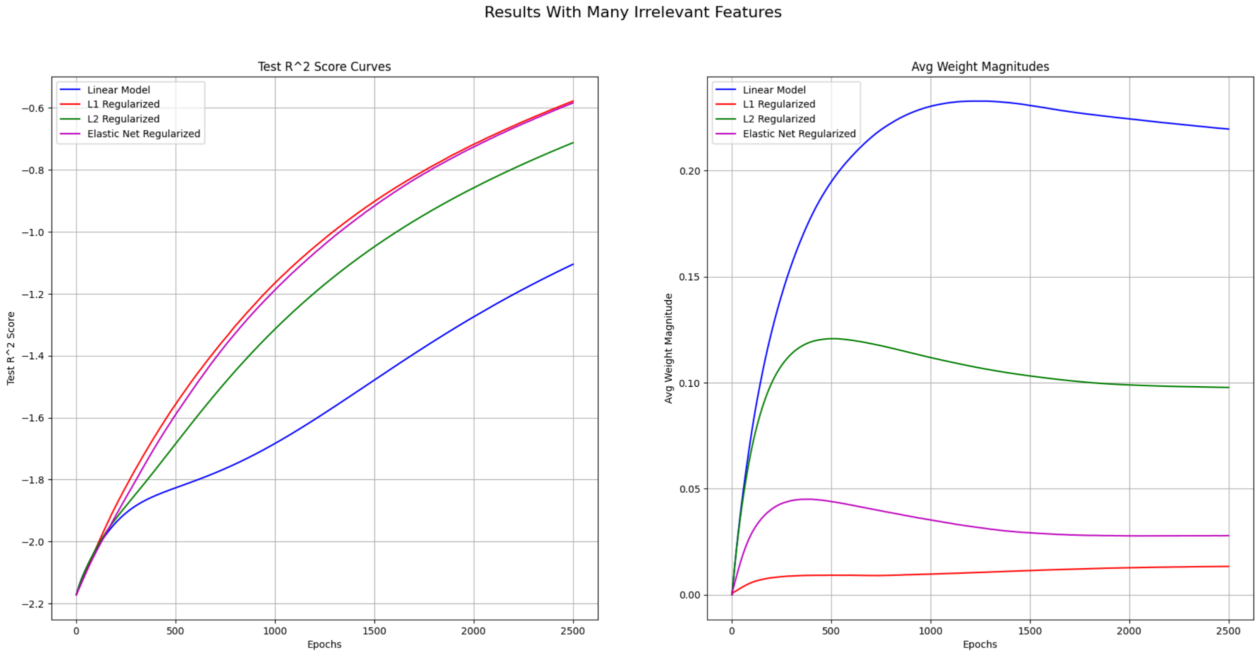
The **Gradient** for this type of **Regularization** is as follows:

$$\nabla f(\beta) = \frac{2}{N}(-X^T)\cdot(Errors) + \alpha((\lambda)sign(\beta_j) + (1-\lambda)\beta_j)$$

Example: Code a **Linear Model** class that can have **L1**, **L2**, or **Elastic Net Regularization** applied. Generate a dataset to use which has mostly irrelevant **Features**. Compute and display the effects on the **Weight Magnitudes (Feature Importance)** and the **Model Performance** (Generation code in the matching notebook).

Output (Feature Importance):

```
Final Linear Model Test R^2:           -1.1051
Final L1 Regularized Model Test R^2:   -0.5784
Final L2 Regularized Model Test R^2:   -0.7128
Final Elastic Net Model Test R^2:      -0.5843
```



Note that the models utilizing **L1 Regularization** showed the best performance on the testing data as they were able to remove many of the irrelevant **Features** from the prediction.

For the above code implementation, an intercept (bias term) was added such that the prediction equation changes to:

$$y_{pred} = \beta_0(1) + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n$$

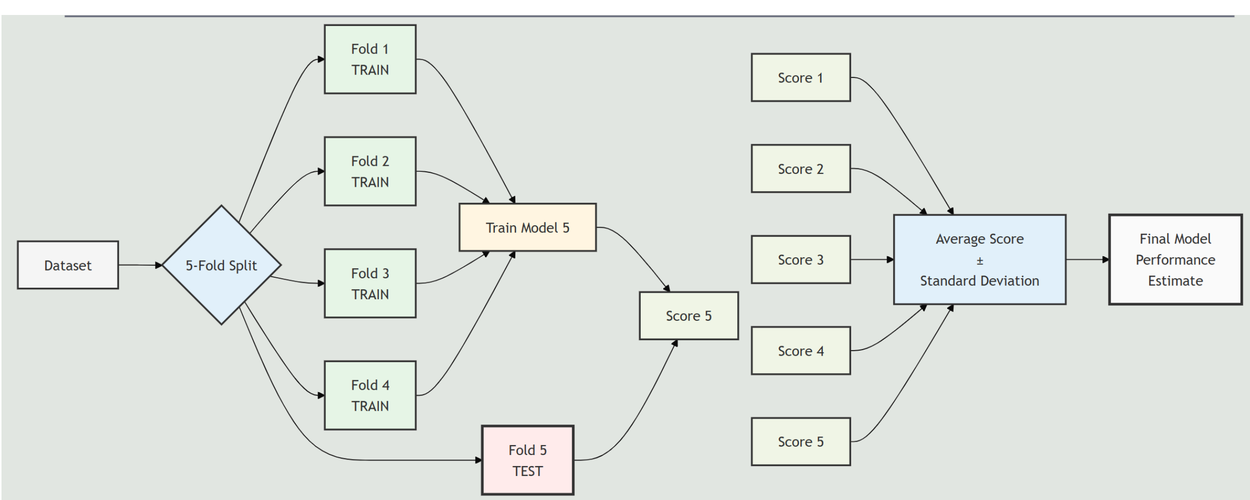
It should be noted that this bias term β_0 roughly corresponds to the mean of the **Target** y. As such, regularization of this term does not make sense as, unlike the other **Weights**, its magnitude does not relate to the model complexity.

3-5 KFold Cross-Validation:

Previously, we discussed the implementation of a **Train-Test Split** to segment our dataset into a **Training** component which the model would be optimized on, and a **Testing** component which would be used to assign the model a final score.

One issue that can emerge with this method is if there is a significant statistical difference between the portion of the data being used for **Training** vs for **Testing**, this is much more common when there are few samples present. This would result in our metrics of score being biased towards models which perform well on our designated **Testing** data.

To address this, we can implement **KFold Cross-Validation**, where the dataset is split into pieces and the model is evaluated multiple times, iterating through which piece is used as the **Testing** dataset:



This removes any preference for a specific segment of the dataset and will be more rigorous than the standard **Train-Test Split**.

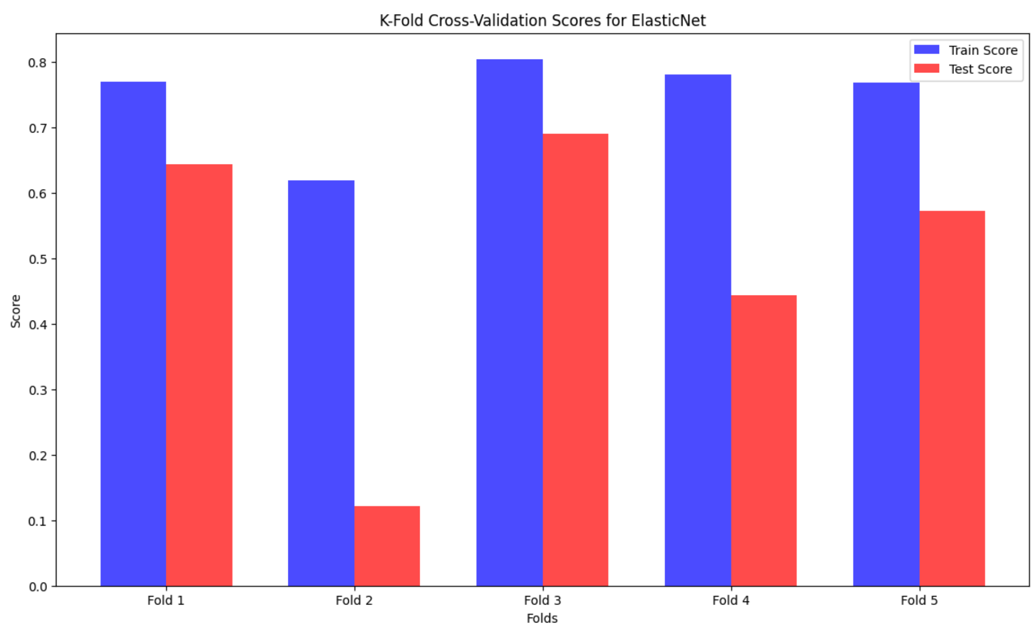
Example: Implement a function which applies **KFold Cross-Validation** to a sklearn model and returns the training and testing scores.

```
from sklearn.linear_model import ElasticNet
from sklearn.preprocessing import StandardScaler

#Define KFold Function
def KFold(model, X, y, k_folds=5):
    #Calculate fold sizes
    N = X.shape[0]
    fold_sizes = (N//k_folds)*np.ones(k_folds, dtype=int)
    fold_sizes[:N % k_folds] += 1 #Distribute remainder

    #Setup to store scores
    train_scores = []
    test_scores = []

    #Loop through folds
    for fold in range(k_folds):
        #Define train/test indices
        test_indices = np.arange(np.sum(fold_sizes[:fold]), np.sum(fold_sizes[:fold + 1]))
        train_indices = np.setdiff1d(np.arange(N), test_indices) #Indices not in test set

        #Split data
        X_train, y_train = X[train_indices], y[train_indices]
        X_test, y_test = X[test_indices], y[test_indices]

        #Standardize features
        scaler = StandardScaler()
        X_train_std = scaler.fit_transform(X_train)
        X_test_std = scaler.transform(X_test)

        #Fit model
        model.fit(X_train_std, y_train)

        #Compute scores
        train_score = model.score(X_train_std, y_train)
        test_score = model.score(X_test_std, y_test)

        #Store scores
        train_scores.append(train_score)
        test_scores.append(test_score)

        #Print fold results
        print(f'Fold {fold + 1}/{k_folds} - Train Score: {train_score:8.4f}, Test Score: {test_score:8.4f}')

    #Print average scores
    print(f'\nAverage Train Score: {np.mean(train_scores):8.4f} +/- {np.std(train_scores):.4f}')
    print(f'Average Test Score:  {np.mean(test_scores):8.4f} +/- {np.std(test_scores):.4f}')

    #Setup for Bar Chart
    labels = [f'Fold {i+1}' for i in range(k_folds)]
    x = np.arange(len(labels))
    width = 0.35
    fig, ax = plt.subplots(figsize=(14, 8))

    #Plot Bar Chart
    ax.bar(x - width/2, train_scores, width, label='Train Score', color='b', alpha=0.7)
    ax.bar(x + width/2, test_scores, width, label='Test Score', color='r', alpha=0.7)
    ax.set_xlabel('Folds')
    ax.set_ylabel('Score')
    ax.set_title(f'K-Fold Cross-Validation Scores for {model.__class__.__name__}')
    ax.set_xticks(x)
    ax.set_xticklabels(labels)
```

```
    ax.legend()
    return train_scores, test_scores, fig, ax


#Generate dataset
X = np.random.rand(25, 5)*10
y = 3*X[:, 0] - 2*X[:, 1]

#Create ElasticNet model
enet_model = ElasticNet(alpha=1.0, l1_ratio=0.5, fit_intercept=True, max_iter=10000)

#Perform K-Fold Cross-Validation
train_scores, test_scores, fig, ax = KFold(enet_model, X, y, k_folds=5)
```

Output:

```
Fold 1/5 - Train Score:   0.7707, Test Score:   0.6445
Fold 2/5 - Train Score:   0.6200, Test Score:   0.1218
Fold 3/5 - Train Score:   0.8044, Test Score:   0.6911
Fold 4/5 - Train Score:   0.7809, Test Score:   0.4442
Fold 5/5 - Train Score:   0.7688, Test Score:   0.5732

Average Train Score:   0.7489 +/- 0.0657
Average Test Score:    0.4949 +/- 0.2044
```



Note that since this dataset has only 25 samples, the differences between the scores of each fold are very high.

3-6 Classification Tasks:

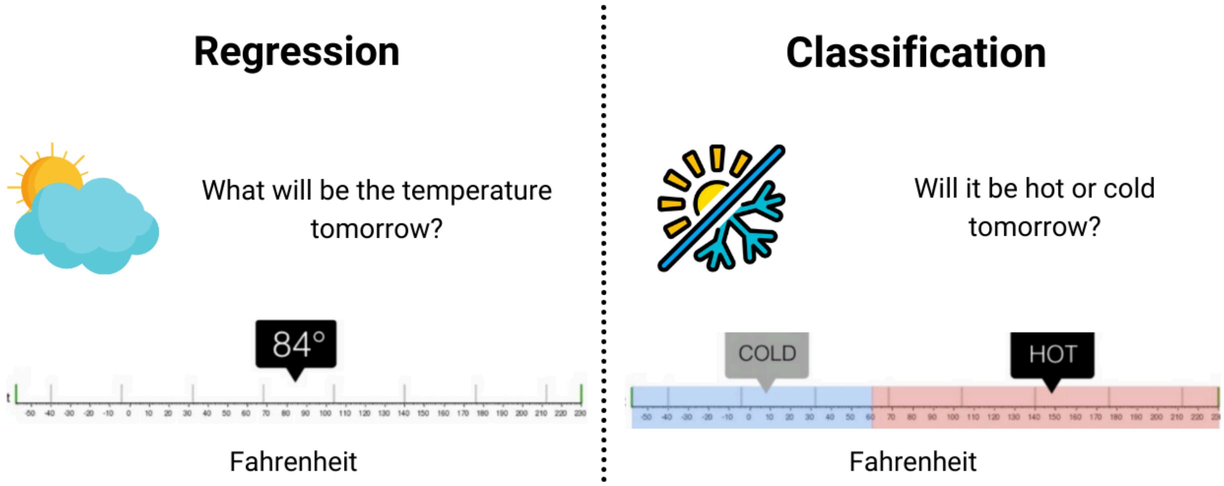
We have been mostly discussing tasks in which the **Target** we are attempting to predict can be any continuous value. Many tasks, however, have **Targets** of which can only take on certain values. These possible values are the **Classes** for the **Target Variable**.

Some examples of these would be:

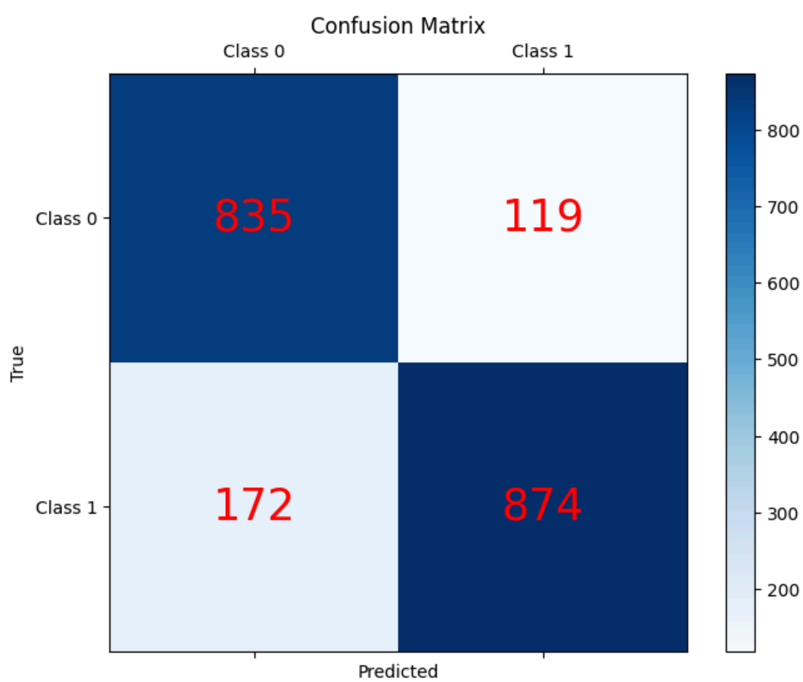
- Predicting whether a candidate receives a job offer:
  - Yes
  - No

- Predicting an student's letter grade:
  - A
  - B
  - C
  - D
  - F

Regression Tasks set out to predict a specific value for the **Target**, whereas **Classification Tasks** set out to predict which **Target's Class**:



When evaluating the performance of a **Classification Model**, it can often be useful to construct a **Confusion Matrix**, which shows all possible combinations of the **Predicted Target Class** and the **True Target Class** as well as how many occurances exist for each combination. I have provided an example below (Generation code in the matching notebook):



Note that the previous **Loss Metrics** we have discussed to score models do not work here. Pretty much all **Loss Metrics** are used for either **Regression** or **Classification** tasks. This also applied to many models, as some have to be written differently depending on the type of task they are assigned. In general, **Classification** metrics will measure some form of **Accuracy** whereas **Regression Metrics** with measure **Errors**.

For now, we will focus on such tasks where there are only two possible values for the **Target**. Such tasks are called **Binary Classification**

3-7 Logistic Regression & Binary Classification:

In order to adapt our basic **Linear Model** to **Binary Classification Tasks**, we can focus on predicting the **Odds Ratio** which describes the model's relative confidence that the **Target Class** is one of the two potential **Classes** (We will encode these as 0 and 1):

$$\frac{P(y_i = 1|\text{feature data})}{P(y_i = 0|\text{feature data})}$$

Note that when dividing two probabilities we will always get a value between $(0 - \infty)$ with higher values indicating a higher chance of the numerator's probability.

We can predict this ratio using **Logistic Regression**, whose prediction equation is as follows:

$$ln\left(\frac{P(y_i = 1|\text{feature data})}{P(y_i = 0|\text{feature data})}\right) = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_n x_{in}$$
$$or$$
$$P(y_i = 1|\text{feature data}) = \frac{1}{1 + e^{-\beta_0 - \beta_1 x_{i1} - \beta_2 x_{i2} - \cdots - \beta_n x_{in}}}$$

Where x_{ij} denotes the jth feature of the ith sample. This equation predicts the probability that the **Target** is the **Primary Class**.

We can assume that the **Target Class** for each sample is decided by:

$$y_i = 1, \text{if} P(y_i = 1|\text{feature data}) \geq 0.5$$
$$\text{and}$$
$$y_i = 0, \text{if} P(y_i = 1|\text{feature data}) < 0.5$$

The **Gradient** used to update the weights for this model is as follows:

$$p_i = P(y_i = 1|\text{weight} = x_i) = \frac{1}{1 + e^{-\beta_0 - \beta_1 x_{i1} - \beta_2 x_{i2} - \cdots - \beta_n x_{in}}}$$

$$\nabla f(\beta_0, \beta) = -\frac{1}{N}\sum_{i=1}^{N}(p_i - y_i)\vec{x}_i$$

This is the **Gradient** of the **Log Loss** or **Binary Cross Entropy (BCE)** function, which is commonly used as a **Loss Function** for **Binary Classification Tasks**.

Typically, when scoring models we also define an **Accuracy** metric where:

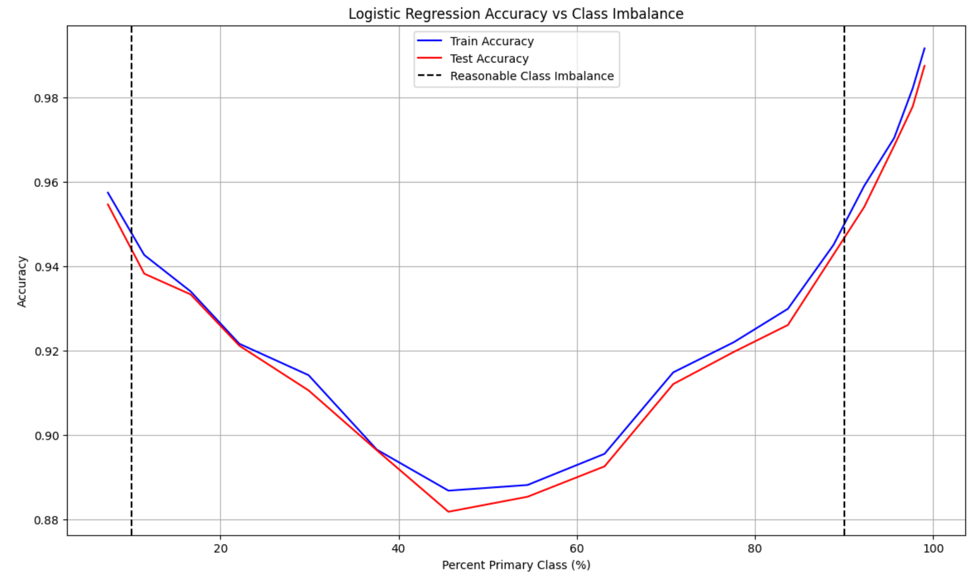
$$\text{Accuracy} = \frac{\sum_{i=1}^{N}(\text{y\_pred}_i == y_i)}{N}$$

This quantifies the percentage of samples for which the model predicted the correct **Target Class**.

Note that this metric can fall apart when the **Distribution of Classes** is not uniform, known as a **Class Imbalance**

Example: Train a **Logistic Regression** model on a **Binary Classification Task** and display the **Train Accuracy** and **Test Accuracy**. Show how this changes as the breakdown of classes changes. (Generation code in the matching notebook).

Output:



Note how the model accuracy increases as the **Class Imbalance** becomes more one-sided. In such cases, **Accuracy** is not a valid measure of performance and a new metric is needed.

Below are some metrics commonly utilized in **Binary Classification**:

Total population (pop.) = 2030	Test outcome **positive**	Test outcome **negative**	Accuracy (ACC) = (TP + TN) / pop. = (20 + 1820) / 2030 ≈ **90.64%**	F_1 score $= 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$ ≈ **0.174**
Actual condition positive	**True positive** (TP) = 20 (2030 × 1.48% × 67%)	**False negative** (FN) = 10 (2030 × 1.48% × (100% − 67%))	True positive rate (TPR), recall, sensitivity = TP / (TP + FN) = 20 / (20 + 10) ≈ **66.7%**	False negative rate (FNR), miss rate = FN / (TP + FN) = 10 / (20 + 10) ≈ **33.3%**
Actual condition negative	**False positive** (FP) = 180 (2030 × (100% − 1.48%) × (100% − 91%))	**True negative** (TN) = 1820 (2030 × (100% − 1.48%) × 91%)	False positive rate (FPR), fall-out, probability of false alarm = FP / (FP + TN) = 180 / (180 + 1820) = **9.0%**	Specificity, selectivity, true negative rate (TNR) = TN / (FP + TN) = 1820 / (180 + 1820) = **91%**
Prevalence = (TP + FN) / pop. = (20 + 10) / 2030 ≈ **1.48%**	Positive predictive value (PPV), precision = TP / (TP + FP) = 20 / (20 + 180) = **10%**	False omission rate (FOR) = FN / (FN + TN) = 10 / (10 + 1820) ≈ **0.55%**	Positive likelihood ratio (LR+) $= \frac{\text{TPR}}{\text{FPR}}$ = (20 / 30) / (180 / 2000) ≈ **7.41**	Negative likelihood ratio (LR−) $= \frac{\text{FNR}}{\text{TNR}}$ = (10 / 30) / (1820 / 2000) ≈ **0.366**
	False discovery rate (FDR) = FP / (TP + FP) = 180 / (20 + 180) = **90.0%**	Negative predictive value (NPV) = TN / (FN + TN) = 1820 / (10 + 1820) ≈ **99.45%**	Diagnostic odds ratio (DOR) $= \frac{\text{LR+}}{\text{LR−}}$ ≈ **20.2**	

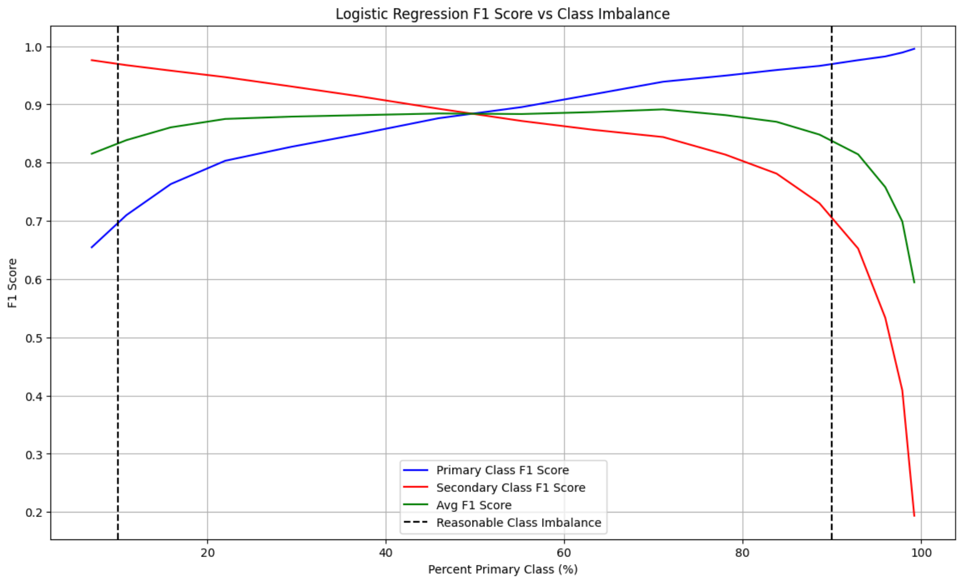
For our purposes, we will use the **F1 Score** to measure the model's objective performance under the class imbalance:

$$\text{F1 Score} = 2\left(\frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}\right)$$

$$\text{Precision} = \frac{TP}{TP + FP}$$

$$\text{Recall} = \frac{TP}{TP + FN}$$

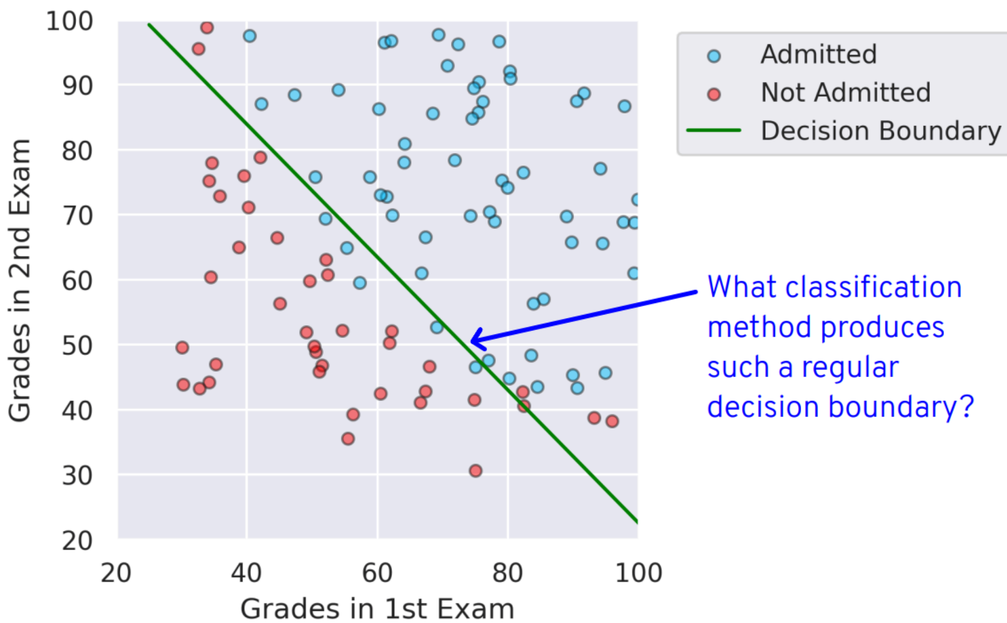
Note that depending on which **Class** is considered "positive" the **F1 Score** is computed differently, so averaging the two versions can yield a **Score Metric** which is robust to **Heavy Class Imbalances**



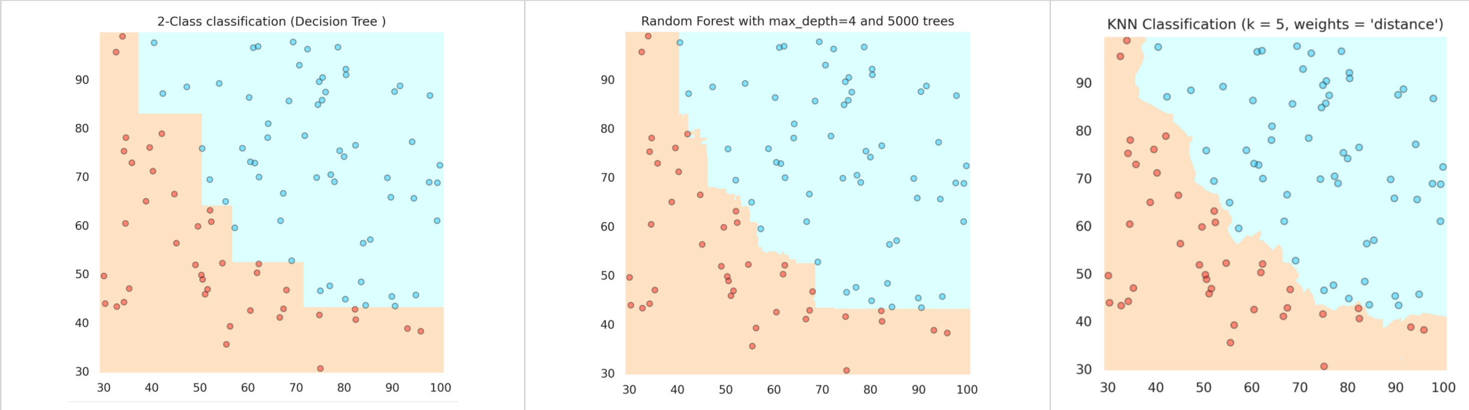
3-8 Multi-Class Classification:

3-N Decision Boundaries & Support Vector Machines (SVM):

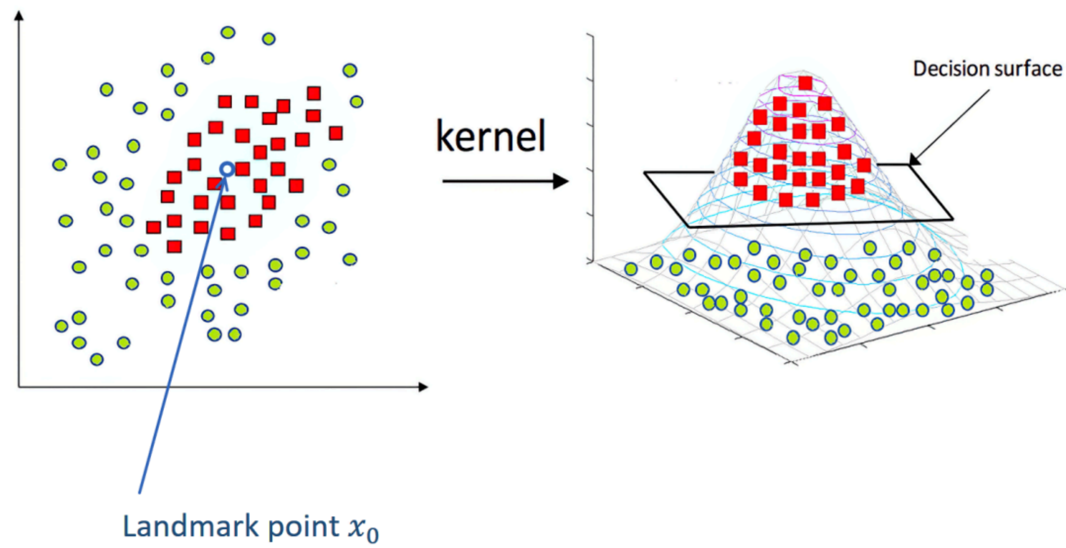
For **Classification Models**, since **Targets** are not continuous, clear boundaries between where the predictions for one class end and another begin must be drawn. These are called **Decision Boundaries** and the type of model used will decide what shapes they can take:



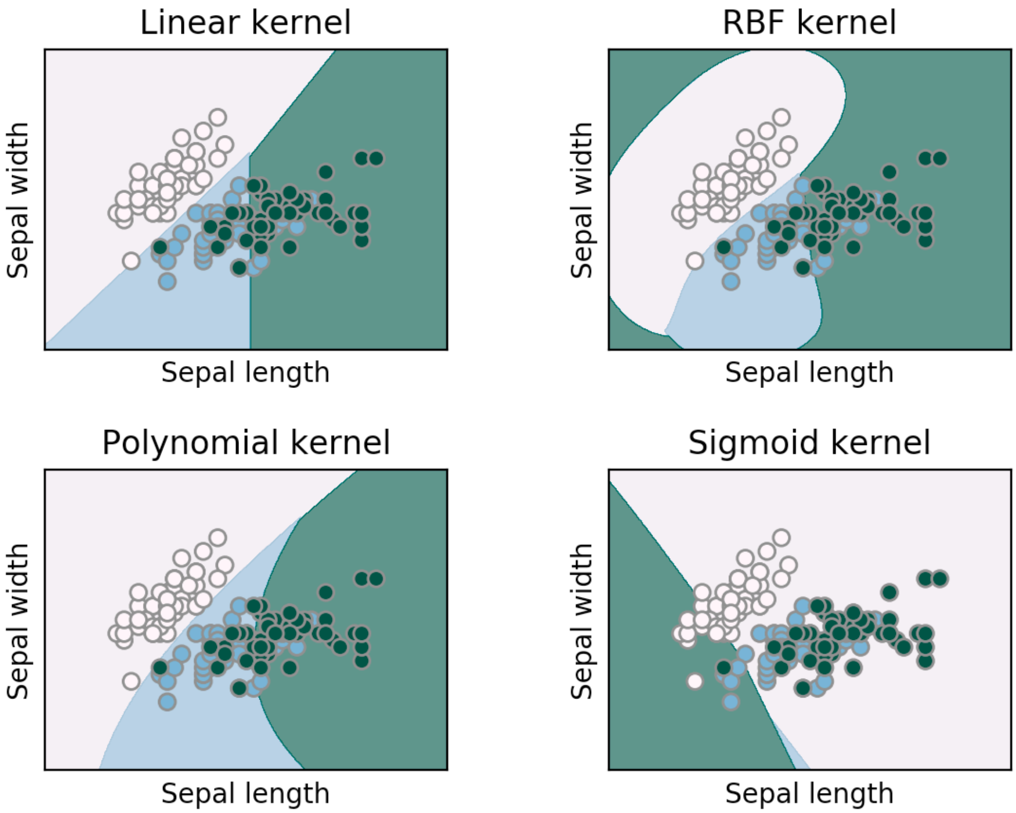
Below are some examples of **Decision Boundaries** created by various models:



We could imagine that we may want a model in which we can engineer the shaping of the **Decision Boundary**. This is done by optimizing the model's accuracy on a set **Kernel** or **Set of Kernels**:



The shape of these **Decision Boundaries** are dictated by the selection of the parameters for the model. This constrained **Kernel** is then optimized to select the shape and size that best divides the different **Target Classes**. Below is an example of how different **Kernel** types can affect the **Decision Boundary**


Linear kernel
RBF kernel
Polynomial kernel
Sigmoid kernel
Sepal width
Sepal length