Luke Waninger
Logistic Regression
Bernoulli to Prediction

Loss function. The standard linear model for regression can be written as the dot product between the independent variables and their respective coefficients: $y_i = \boldsymbol{x_i}^T\boldsymbol{\beta}$. Finding the probability of an input sample being positively classified is performed by adding the logit link function to generate a general linear model. As such, the probability of a positive classification conditioned on the input sample can be written as:

$$P(Y = 1 \,|\, x_i\boldsymbol{\beta}) = \frac{e^{x_i^T\boldsymbol{\beta}}}{1 + e^{x_i^T\boldsymbol{\beta}}} \tag{1}$$

However, this transforms the output to a probability between 0 and 1. For ML purposes, I transform the link function's range from [0, 1] to [-1, 1].

$$P(Y = 1 \,|\, x_i\boldsymbol{\beta}) = \frac{2e^{x_i^T\boldsymbol{\beta}}}{1 + e^{x_i^T\boldsymbol{\beta}}} - 1 \tag{2}$$

Classifying any input sample has a binary output so can be modeled via a Bernoulli distribution. We can therefore write liklihood of a positive classification as:

$$\begin{aligned}
\mathcal{L}(\boldsymbol{\beta}) &= \Pi_{i=1}^n P(y_i = 1|\boldsymbol{x_i\beta})^{y_i} P(y_i = -1|\boldsymbol{x_i\beta})^{1-y_i} \\
&= \Pi_{i=1}^n (1 - P(y_i = 1|\boldsymbol{x_i\beta}))^{1-y_i} P(y_i = 1|\boldsymbol{x_i\beta})^{y_i}
\end{aligned}$$

Replacing with equation 2 we find the liklihood and log-liklihood to be:

$$(1 - P(y_i = 1|\boldsymbol{x_i\beta}))^{1-y_i} P(y_i = 1|\boldsymbol{x_i\beta})^{y_i} = \Big(\frac{2e^{\boldsymbol{x_i\beta}}}{1 + e^{\boldsymbol{x_i\beta}}} - 1\Big)^{1-y_i} 2^{y_i}\Big(1 - \frac{e^{\boldsymbol{x_i\beta}}}{1 + e^{\boldsymbol{x_i\beta}}}\Big)^{y_i}$$

$$\begin{aligned}
\mathcal{L}(\boldsymbol{\beta}) &= \prod_{i=1}^n \Big(\frac{2e^{\boldsymbol{x_i\beta}} - (1 + e^{\boldsymbol{x_i\beta}})}{1 + e^{\boldsymbol{x_i\beta}}}\Big)^{1-y_i} 2^{y_i}\Big(\frac{1}{1 + e^{\boldsymbol{x_i\beta}}}\Big)^{y_i} \\
&= \prod_{i=1}^n \Big(\frac{e^{\boldsymbol{x_i\beta}} - 1}{1 + e^{\boldsymbol{x_i\beta}}}\Big)^{1-y_i} 2y_i\Big(\frac{1}{1 + e^{\boldsymbol{x_i\beta}}}\Big)^{y_i} \\
\text{log}(\mathcal{L}(\boldsymbol{\beta})) &= \sum_{i=1}^n (1 - y_i)\text{log}\Big(\frac{e^{\boldsymbol{x_i\beta}} - 1}{1 + e^{\boldsymbol{x_i\beta}}}\Big) 2y_i \text{log}(1 + e^{-y_i\boldsymbol{x_i\beta}}) \\
&= \sum_{i=1}^n \text{log}(1 + e^{-y_i\boldsymbol{x_i\beta}})
\end{aligned}$$

Next we normalize, add in an L_2 norm, and minimize the negative logliklihood to obtain our objective function.

$$F(\boldsymbol{\beta}) = \min_{\boldsymbol{\beta}\in\Re^d} \frac{1}{n}\sum_{i=1}^n \text{log}(1 + e^{-y_i\boldsymbol{x_i\beta}}) + \lambda||\boldsymbol{\beta}||^2 \tag{3}$$

The objective function implemented in Python is fairly straight forward.

```python
def __objective(self, beta):
    """calculate objective value

    Args:
        beta: dX1 (ndarray) weight coefficients

    Returns:
        float
    """
    x, y, n, l = self.__x, self.__y, self.__n, self._lamda

    loss = np.sum([np.log(1 + np.exp(-yi*xi.T@beta)) for xi, yi in zip(x, y)])
    return loss/n + l*np.linalg.norm(beta)**2
```

Gradient descent. In order to optimize the loss function, I choose to use gradient descent; reversing the derivative to descend toward the global minimum. To do this, I take advantage of the linearity of differentiation and begin by assuming both n and d are equal to one. Then, the loss function becomes:

$$
\begin{aligned}
F(\boldsymbol{\beta}) &= \frac{1}{1}\sum_{i=1}^{1} \log(1 + e^{-y_1 \boldsymbol{x_1}\boldsymbol{\beta}}) + \lambda||\boldsymbol{\beta}||^2 \\
&= \log(1 + e^{-y_1 \boldsymbol{x_1}\boldsymbol{\beta}}) + \lambda||\boldsymbol{\beta}||^2 \\
\frac{d}{d\boldsymbol{\beta}}F(\boldsymbol{\beta}) &= \frac{1}{(1 + e^{-y_1 \boldsymbol{x_1}\boldsymbol{\beta}})}\log(1 + e^{-y_1 \boldsymbol{x_1}\boldsymbol{\beta}}) + 2\lambda\boldsymbol{\beta} \\
&= \frac{-y_1 \boldsymbol{x_1} e^{-y_1 x_1 \boldsymbol{\beta}}}{1 + e^{-y_1 \boldsymbol{x_1}\boldsymbol{\beta}}} + 2\lambda\boldsymbol{\beta}
\end{aligned}
$$

Now, we assume n and d are greater than 1, and using equation 1 (as p) in line 2

$$
\begin{aligned}
&= \frac{1}{n}\left[\frac{-y_1 \boldsymbol{x_1} e^{-y x_1 \boldsymbol{\beta}}}{1 + e^{-y_1 \boldsymbol{x_1}\boldsymbol{\beta}}} + \cdots + \frac{-y_n \boldsymbol{x_n} e^{-y x_n \boldsymbol{\beta}}}{1 + e^{-y_n \boldsymbol{x_n}\boldsymbol{\beta}}}\right] + 2\lambda\boldsymbol{\beta} \\
&= -\frac{1}{n}\Big[-y_1 \boldsymbol{x_1} p_1 + \cdots + -y_n \boldsymbol{x_n} p_n\Big] + 2\lambda\boldsymbol{\beta} \\
&= -\frac{1}{n}\sum_{i=1}^{n} y_i \boldsymbol{x_i} p_i + 2\lambda\boldsymbol{\beta}
\end{aligned}
$$

$$
\frac{d}{d\boldsymbol{\beta}}F(\boldsymbol{\beta}) = -\frac{1}{n}Y^T P X + 2\lambda\boldsymbol{\beta} \quad \text{where} \quad X \in \Re^{nXd}, P \in \Re^{nXn}, Y \in \Re^{1Xn} \tag{4}
$$

Again, writing equation in Python is incredibly straight forward.

```python
def __gradient(self, b):
    """calculate gradient

    Args:
        b: dX1 (ndarray) weight coefficients

    Returns:
        dX1 ndarray gradient
```

```python
        """
        x, y, l, n = self.__x, self.__y, self._lamda, self.__n

        p = (1 + np.exp(y*(x @ b)))**-1
        return 2*l*b - (x.T @ np.diag(p) @ y)/n
```

Optimization is performed iteratively using the fast-gradient descent algorithm and back-tracking line-search (Armijo stopping condition) for finding the optimal learning rate. Proofs of these algorithms are not including in this document but, the Python code is shown below. It can be reused for any optimization technique that allows for gradient descent.

```python
    def fit(self, x_train, y_train, pos=1, neg=-1, eta=None, queue=None):
        """fit the classifier

        Args:
            x_train: nXd (ndarray) of training samples
            y_train: nX1 (ndarray) of true labels
            pos: (object) positive class label
            neg: (object) optional negative class label
            eta: (float) optional learning rate
            queue: (Queue) optional logging queue
        Returns:
            trained classifier
        """
        self._pos, self._neg = pos, neg

        self.__x, self.__y = x_train, y_train
        self.__n, self.__d = x_train.shape
        self._eta = self.__calc_t_init() if eta is not None else eta

        self.__log_queue = queue
        self.__fgrad()

        self.__log(dict(
            klass=self._pos,
            iter='END',
            obj=self.__objective(self.coef),
            coef=self.coef)
        )
        return self

    def __backtracking(self, beta, t_eta=0.5, alpha=0.5):
        """backtracking line search

        Args:
            beta: dX1 (ndarray) weight coefficients
            t_eta: (float) optional [default=0.5] 0 < t_eta < 1 learning rate for eta
            alpha: (float) optional [default=0.5] 0 < alpha < 1 tune stopping condition

        Returns:
            float: optimum learning rate
        """
        l, t = self._lamda, 1

        gb = self.__gradient(beta)
```

```python
        n_gb = np.linalg.norm(gb)

        found_t, i = False, 0
        while not found_t and i < 100:
            if self.__objective(beta - t*gb) < self.__objective(beta) - alpha*t*n_gb**2:
                found_t = True
            elif i == self._max_iter-1:
                break
            else:
                t *= t_eta
                i += 1

        self.__eta = t
        return self.__eta
```

Prediction. We can take advantage of the following proof to simplify the prediction calculations. We begin by showing that $P(Y=1|\boldsymbol{\beta}) > P(Y=-1|\xi\boldsymbol{\beta}) \iff \xi^T\boldsymbol{\beta}^* > 0$

$$P(Y=1|\boldsymbol{x_i}\boldsymbol{\beta}) = \frac{e^{\boldsymbol{x}_i^T\boldsymbol{\beta}}}{1+e^{\boldsymbol{x}_i^T\boldsymbol{\beta}}}$$

$$= \frac{1}{1+e^{\boldsymbol{x}_i^T\boldsymbol{\beta}}}$$

We need to determine when the ratio of probabilities is > 1.

$$\frac{P(Y=1|\boldsymbol{x_i}\boldsymbol{\beta})}{P(Y=-1|\boldsymbol{x_i}\boldsymbol{\beta})} = \frac{e^{\boldsymbol{x}_i^T\boldsymbol{\beta}}}{1+e^{\boldsymbol{x_i}\boldsymbol{\beta}}}(1+e^{\boldsymbol{x}_i^T\boldsymbol{\beta}})$$

$$\text{therefore} \quad e^{\boldsymbol{x}_i^T\boldsymbol{\beta}} > 0 \iff \boldsymbol{x_i}^T\boldsymbol{\beta} > 0$$

Which gives the following Python snippet for classifying a new sample.

```python
    def predict(self, x, beta=None):
        """prediction probabilities

        Args:
            x: nXd (ndarray) of input values
            beta: dX1 (ndarray) optional weight coefficients
        Returns:
            nX1 ndarray of class labels
        """
        beta = self.coef if beta is None else beta
        return [self._pos if xi @ beta > 0 else self._neg for xi in x]
```