1 Linear regression: regularization and logistic regression

Today we will look at two very useful extensions of linear regression algorithms.

In [1]:
```
import random

import numpy as np
import pandas as pd
from scipy import stats
import matplotlib
import matplotlib.pyplot as plt
from sklearn import metrics
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn import linear_model
from sklearn.linear_model import Ridge, Lasso, LogisticRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import cross_validate, cross_val_predict, KFold

font = {'size'   : 16}
matplotlib.rc('font', **font)
matplotlib.rc('xtick', labelsize=14)
matplotlib.rc('ytick', labelsize=14)
matplotlib.rcParams.update({'figure.autolayout': False})
matplotlib.rcParams['figure.dpi'] = 100
```

1.1 Step 1: Generate Data

First let's generate a dataset with more than one feature variable to explore algorithms that can be used to reduce overfitting.

In [2]:
```
np.random.seed(16) #set seed for reproducibility purposes

x1 = np.arange(100)

x2 = np.linspace(0,1,100)

x3 = np.logspace(2,3,num=100)

ypb = 3*x1 + 0.5*x2 + 15*x3 + 3 + 5*(np.random.poisson(3*x1 + 0.5*x2 + 15*x3,100)-(3*x1 + 0.5*x2 + 15*x3))
                                                    #generate some data with scatter following Poisson distribution
                                                    #with exp value = y from linear model, centered around 0

xb = np.vstack((x1,x2,x3)).T # this gives us three features
```

Now we add correlated features with a polynomial transformation:

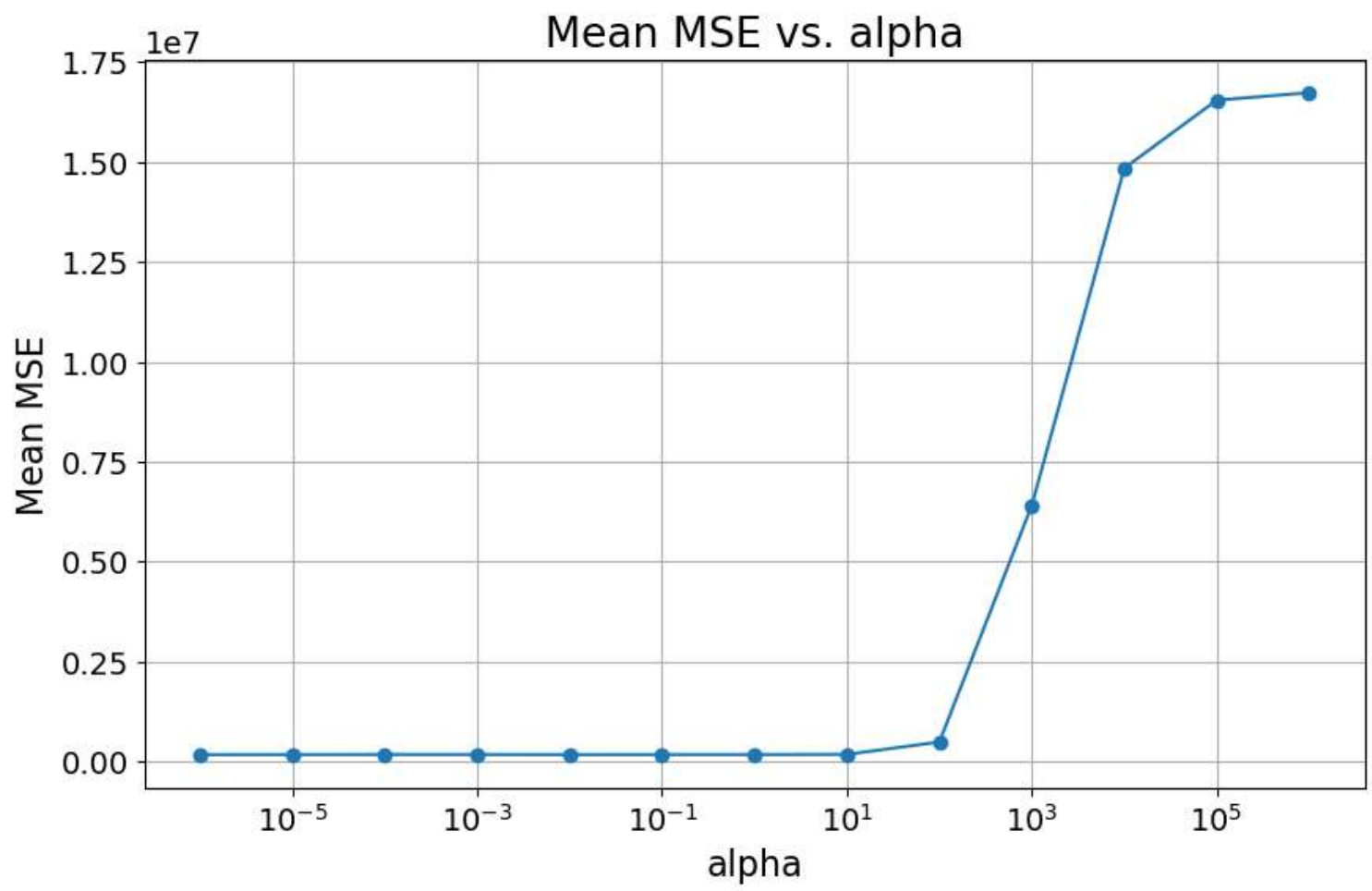
In [3]:
```
poly = PolynomialFeatures(2, include_bias=False)

new_xb = poly.fit_transform(xb) # this dives us 9 features total

new_xb.shape
```

Out[3]: (100, 9)

1.2 Step 2: Ridge regression

Set up Ridge regression, and determine cross-validated scores for different values of alpha that are logarithmically spaced in the rage between 10^{-6} and 10^{6}, then find the best alpha that optimizes the test score. For this to be meaningful, the data should be standardized, so please set up a pipeline with Ridge() and the StandardScaler() you used earlier. Make a plot of mean test scores versus alpha. Which scale is more appropriate, linear or logarithmic?

In [7]:
```
from sklearn.pipeline import Pipeline
from sklearn.model_selection import train_test_split, cross_val_score

X_train, X_test, y_train, y_test = train_test_split(new_xb, ypb, test_size=0.2, random_state=42)

alphas = np.logspace(-6, 6, num=13)

pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('ridge', Ridge())
])

cv_scores_mean = []

for alpha in alphas:
    pipeline.set_params(ridge__alpha=alpha)
    scores = cross_val_score(pipeline, X_train, y_train, cv=10, scoring='neg_mean_squared_error')
    cv_scores_mean.append(-np.mean(scores))

plt.figure(figsize=(10, 6))
plt.plot(alphas, cv_scores_mean, marker='o')
plt.xlabel('alpha')
plt.ylabel('Mean MSE')
plt.title('Mean MSE vs. alpha')
plt.xscale('log')
plt.grid()
plt.show()
```



In [8]:
```
print(
f"""
A logarithmic Scale is more appropriate
""")
```

```
A logarithmic Scale is more appropriate
```

There is a built-in routine for this hyperparameter optimization scan called RidgeCV. Try it out, do you get the same answer as before?

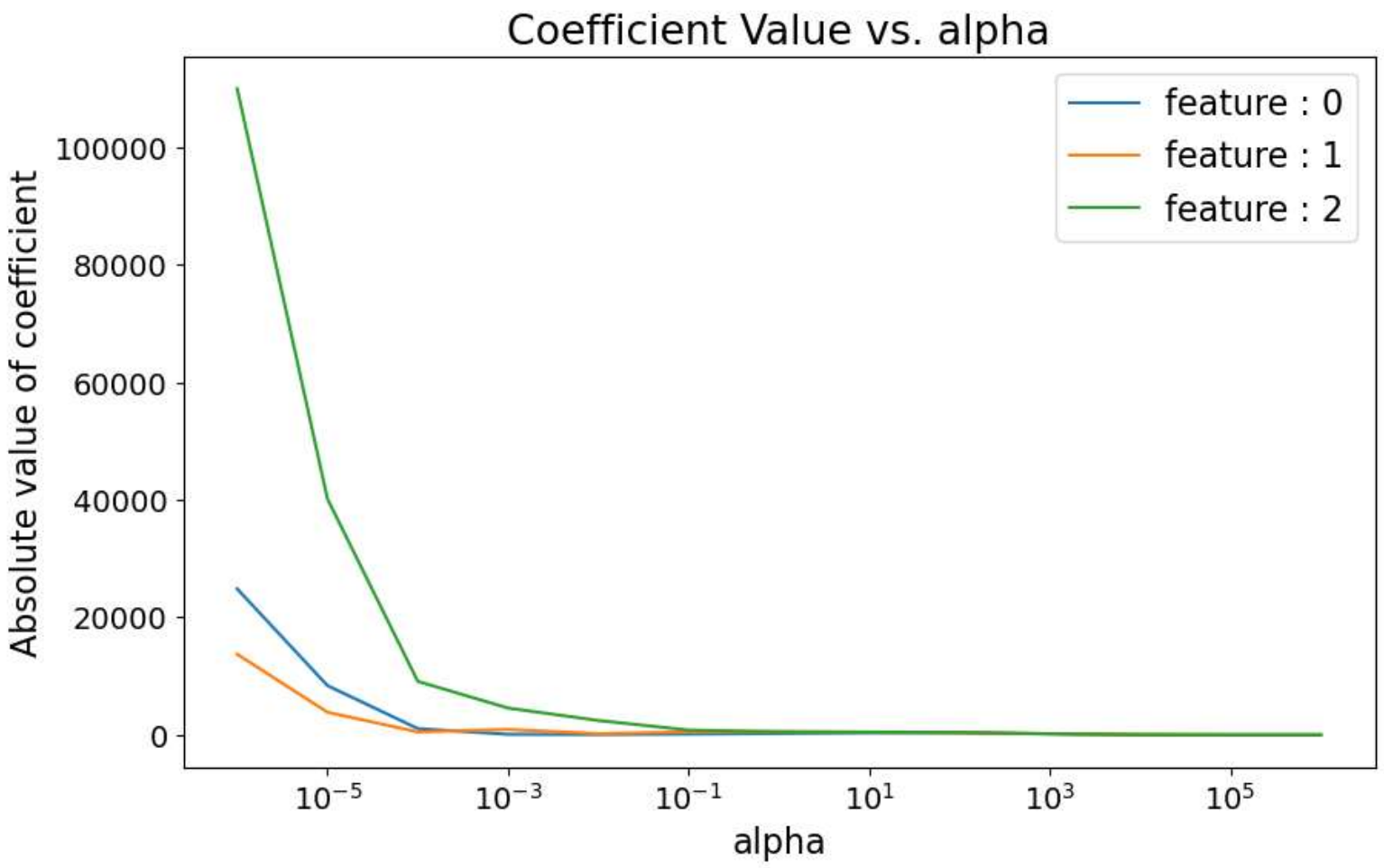
In [10]:
```
from sklearn.linear_model import RidgeCV

alphas = np.logspace(-6, 6, num=13)
pipeline_cv = Pipeline([
    ('scaler', StandardScaler()),
    ('ridgecv', RidgeCV(alphas=alphas, store_cv_values=True))
])
pipeline_cv.fit(X_train, y_train)
best_alpha = pipeline_cv.named_steps['ridgecv'].alpha_
cv_values = pipeline_cv.named_steps['ridgecv'].cv_values_
mean_mse_for_best_alpha = cv_values.mean(axis=0)[np.where(alphas == best_alpha)]

best_alpha, mean_mse_for_best_alpha
```

Out[10]:
```
(1e-06, array([175721.88208223]))
```

The coefficients of the linear model are strongly affected by regularization. Pick one or two of the features and plot the absolute value of the coefficient for the range of alphas used above in the search scan. Which scale makes most sense for this plot, linear or logarithmic?

Remember that the data must be standardized for this comparison to make sense, so either use a pipeline or fit_transform your data first to standard form using StandardScaler.

In [31]:
```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
coefficients = []

for alpha in alphas:
    ridge = Ridge(alpha=alpha)
    ridge.fit(X_train_scaled, y_train)
    coefficients.append(ridge.coef_)

coefficients = np.array(coefficients)

n_prompt = 3
column_indices = np.arange(new_xb.shape[1])
np.random.shuffle(column_indices)
selected_features = new_xb[:, column_indices[:n_prompt]]

selected_coefficients = coefficients[:,column_indices[:n_prompt]]

plt.figure(figsize=(10, 6))
for i in range(selected_features.shape[1]):
    plt.plot(alphas, np.abs(selected_coefficients[:, i]), label=f"feature : {i}")

plt.xlabel('alpha')
plt.ylabel('Absolute value of coefficient')
plt.title('Coefficient Value vs. alpha')
plt.xscale('log')
plt.legend()
plt.show()
```



1.3 Step 3: Lasso regression

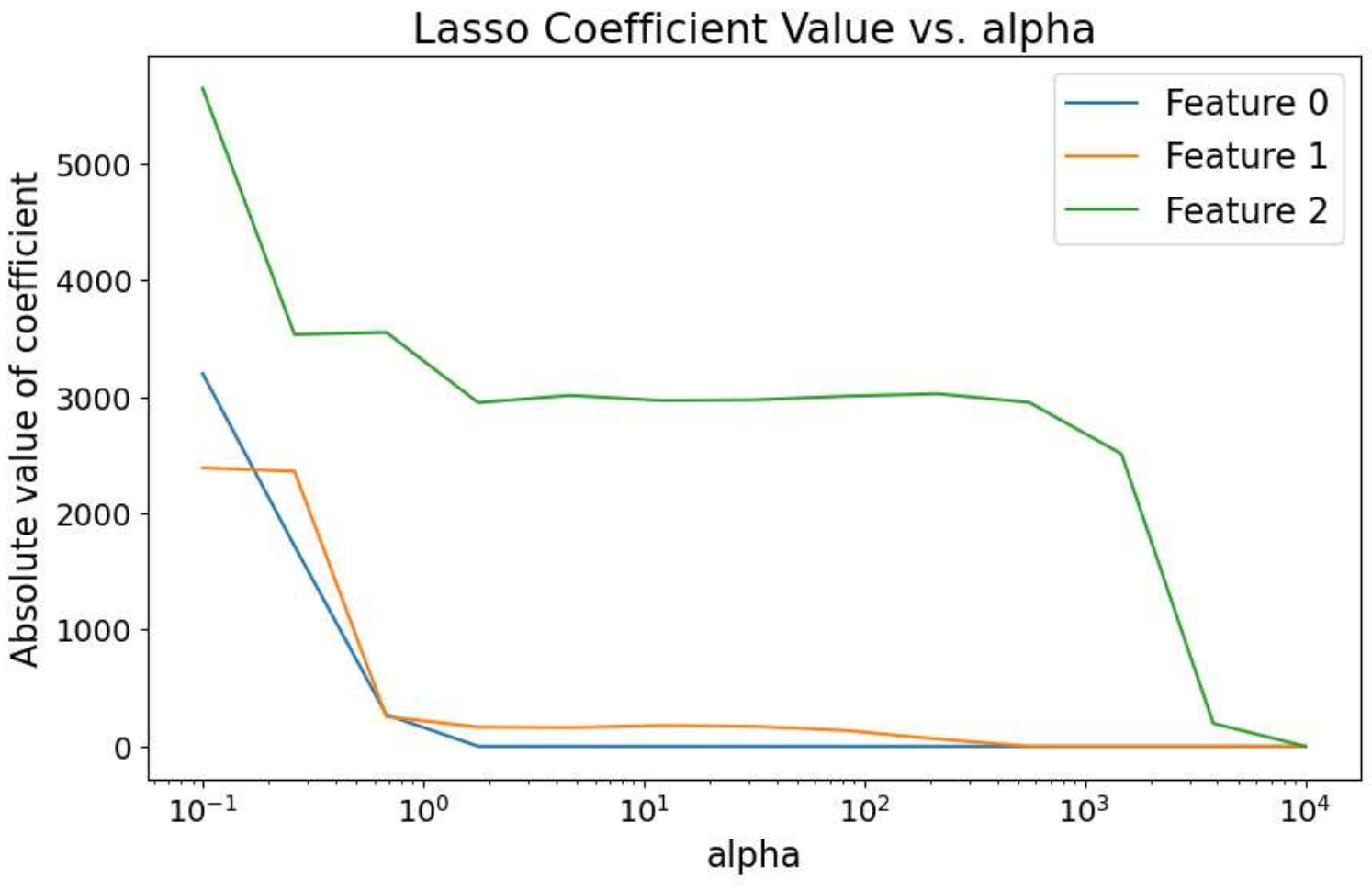
Repeat the items of step 2 for Lasso regression, now for a smaller range of alpha values 10^{-1} and 10^{4}. Also, to avoid numerical instabilities, in Lasso set the parameters *max_iter = 10000000, tol = 1e-6*

In [35]:
```
X_train, X_test, y_train, y_test = train_test_split(new_xb, ypb, test_size=0.2, random_state=42)

alphas = np.logspace(-1, 4, num=13)
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('lasso', Lasso())
])
cv_scores_mean = []
for alpha in alphas:
    pipeline.set_params(lasso__alpha=alpha)
    scores = cross_val_score(pipeline, X_train, y_train, cv=10, scoring='neg_mean_squared_error')
    cv_scores_mean.append(-np.mean(scores))

plt.figure(figsize=(10, 6))
plt.plot(alphas, cv_scores_mean, marker='o')
plt.xlabel('alpha')
plt.ylabel('Mean MSE')
plt.title('Mean MSE vs. alpha for Lasso Regression')
plt.xscale('log')
plt.grid(True)
plt.show()

print(
f"""
Answer:
-----------------------
It Appears a linear scale would work reasonably well for lasso as the error changes mostly in the $$10^2$$ to $$10^3$$ range
""")
```

```
C:\Users\kesha\miniconda3\envs\cs425\lib\site-packages\sklearn\linear_model\_coordinate_descent.py:628: ConvergenceWarning: Objective did not converge. You might want to increase the number of iterations, check the scale of the features or consider increasing regularisation. Duality gap: 3.377e+06, tolerance: 1.159e+05
  model = cd_fast.enet_coordinate_descent(
C:\Users\kesha\miniconda3\envs\cs425\lib\site-packages\sklearn\linear_model\_coordinate_descent.py:628: ConvergenceWarning: Objective did not converge. You might want to increase the number of iterations, check the scale of the features or consider increasing regularisation. Duality gap: 4.554e+06, tolerance: 1.152e+05
  model = cd_fast.enet_coordinate_descent(
C:\Users\kesha\miniconda3\envs\cs425\lib\site-packages\sklearn\linear_model\_coordinate_descent.py:628: ConvergenceWarning: Objective did not converge. You might want to increase the number of iterations, check the scale of the features or consider increasing regularisation. Duality gap: 4.436e+06, tolerance: 1.180e+05
  model = cd_fast.enet_coordinate_descent(
C:\Users\kesha\miniconda3\envs\cs425\lib\site-packages\sklearn\linear_model\_coordinate_descent.py:628: ConvergenceWarning: Objective did not converge. You might want to increase the number of iterations, check the scale of the features or consider increasing regularisation. Duality gap: 4.699e+06, tolerance: 1.205e+05
  model = cd_fast.enet_coordinate_descent(
C:\Users\kesha\miniconda3\envs\cs425\lib\site-packages\sklearn\linear_model\_coordinate_descent.py:628: ConvergenceWarning: Objective did not converge. You might want to increase the number of iterations, check the scale of the features or consider increasing regularisation. Duality gap: 4.570e+06, tolerance: 1.201e+05
```

In [42]:
```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
coefficients = []

for alpha in alphas:
    lasso = Lasso(alpha=alpha, max_iter=10000)
    lasso.fit(X_train_scaled, y_train)
    coefficients.append(lasso.coef_)

coefficients = np.array(coefficients)

n_prompt = 3
column_indices = np.arange(new_xb.shape[1])
np.random.shuffle(column_indices)
selected_features = new_xb[:, column_indices[:n_prompt]]

selected_coefficients = coefficients[:,column_indices[:n_prompt]]

plt.figure(figsize=(10, 6))
for i in range(selected_features.shape[1]):
    plt.plot(alphas, np.abs(selected_coefficients[:, i - column_indices[0]]), label=f"Feature {i}")

plt.xlabel('alpha')
plt.ylabel('Absolute value of coefficient')
plt.title('Lasso Coefficient Value vs. alpha')
plt.xscale('log')
plt.legend()
plt.show()
```

```
C:\Users\kesha\miniconda3\envs\cs425\lib\site-packages\sklearn\linear_model\_coordinate_descent.py:628: ConvergenceWarning: Objecti
ve did not converge. You might want to increase the number of iterations, check the scale of the features or consider increasing re
gularisation. Duality gap: 1.353e+06, tolerance: 1.311e+05
  model = cd_fast.enet_coordinate_descent(
C:\Users\kesha\miniconda3\envs\cs425\lib\site-packages\sklearn\linear_model\_coordinate_descent.py:628: ConvergenceWarning: Objecti
ve did not converge. You might want to increase the number of iterations, check the scale of the features or consider increasing re
gularisation. Duality gap: 1.961e+05, tolerance: 1.311e+05
  model = cd_fast.enet_coordinate_descent(
```



Type *Markdown* and LaTeX: α^2

In [ ]:

1.4 Step 4: Logistic regression

Logistic regression can be used as a classifier on categorical data. To check it out, let's revisit the habitable planet data set from Lab 3, *HPLearningSet.csv*.

Load the data set and create a feature matix X with the first two features, mass and period. Fit sklearn's *LogisticRegression(random_state=1,penalty=None)*, print the prediction and the accuracy for this model.

In [ ]:

In [ ]:

Now we will try to create our own implementation of logistic regression, using gradient descent. The relevant loss function is the negative of eq. (5.25). First, you need to compute analytically the gradient of the loss function wrt. β. This replace the MSE loss in your gradient descent code from lab 10. To facilitate the code, it is useful to also define the sigmoid function eq. (5.21) that returns the probabilities that you will need in the gradient. You could then define another function *predict* that takes the probabilities and returns either 0 or 1, depending on whether the probabilities are less than or greater than 0.5.

We can set *np.random.seed(1)* to intialize a suitable solution vector. Remember that a column of ones must be added to the feature matrix X just as before to take care of the offset. Try starting with a learning rate of $\eta = 0.0001$.

Print the accuracy and predictions of your own logistic regression and compare to sklearn's answer. You might need a quite large number of iterations (up to 10^6) to reach the same predictions.

In [ ]:

In [ ]: