Problem 1.

1.

[x,y] = [-2,3]

This problem is neither underdetermined or overdetermined, it has the exact number of equations for the number of unknowns.

2.

[x,y] = [-2,3]

Overdetermined, because there are more equations than unknowns.

3.

Least squares cannot be used in a underdetermined problem, because there are infinitely many solutions. So, the error would be 0.

4.

Least squares is designed to solve for the best solution in an overdetermined problem, and Least squares happens to be convex. The error could be zero, like the case in 3.1.2 but that is because it is a special case where there is a perfect solution for all three matrices. Most of the time, there will be some residual error, that depends on the variance in the system.

Problem 2.

1.

The least squares problem is a regression method to determine a solution for an overdetermined data set. Essentially, when determining a regression line, the goal is to minimize the squares of the residuals, so you end up with the “least squares”. What this does, is it weights outliers more, because large residuals get squared and are favored. Least squares is favored because of its relationship to the natural gaussian distribution. If errors are normal distributed among the regression line, then the least squares fit is considered optimal. So, least square fit can work well in many real-life scenarios because there are a lot of cases where the noise (error) is normal distributed.

2. <http://mathworld.wolfram.com/LeastSquaresFitting.html>

R is the residual (distance from regression line to the data point)

 R^2=sum[y_i-f(x_i,a_1,a_2,...,a_n)]^2 

The condition for R^2 to be a minimum is that

|  |
| --- |
| (partial(R^2))/(partiala_i)=0 |

For a linear fit,

 f(a,b)=a+bx, 

So

|  |
| --- |
| R^2(a,b)=sum_(i=1)^n[y_i-(a+bx_i)]^2 |
| (partial(R^2))/(partiala)=-2sum_(i=1)^n[y_i-(a+bx_i)]=0 |

|  |
| --- |
| (partial(R^2))/(partialb)=-2sum_(i=1)^n[y_i-(a+bx_i)]x_i=0. |

3.

[a,b] = [-2,3]

4.

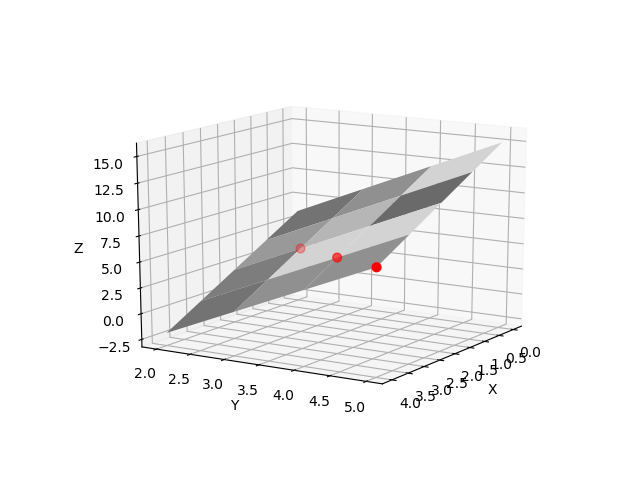
The equations are similar because x and y in problem 1 have the same solution as a and b. But instead of finding data that fits a system of equations, we are finding the weights that best fit a set of data points.

5.

Yes. Solved using least squares in python linear algebra package.

[a,b] = [-2,3]

6.



7.

Error = 4.93038066e-32 ~= 0

Problem 3

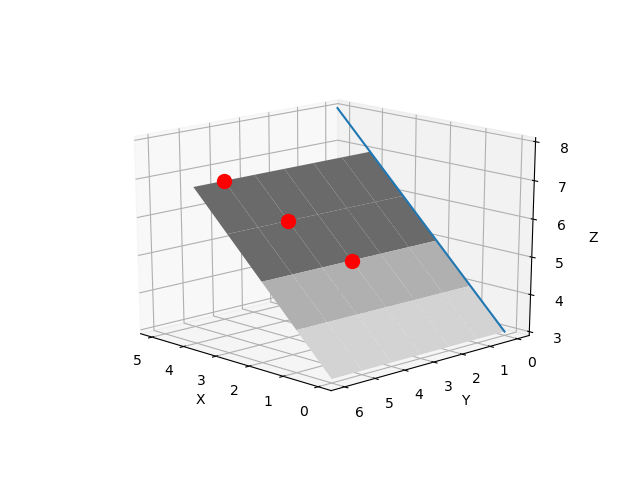
1.

2.

[a1,a2,a3] = [-5.672756082607037e-16 1.0000000000000033 2.999999999999999]

~= [0,1,3]

3.



4.

Error = 0

5.

Both functions fit the data perfectly. Because the three points are on a line (there is one unique linear solution), there are an infinite number of linear 3d solutions. In this problem we did not account for the Y component of each data point, but we could also get an error free solution if we did not account for the X component. By eliminating the Y component from our solution, we are finding a solution a Y plane.

6.

We can use any function we can imagine for **ϕ.**

Problem 4.

1.

Regularization introduces new information to prevent overfitting. L1 and L2 norm are forms of regularization. Regularization reduces variance in a regression model, without losing too much bias. It can also reduce factoring the outliers (noise) into the model and can reduce the model space (lasso).

2.

No. we did not use regularization. By using least squares we actually weighted the outliers (noise) heavier. We did not add any new information to our data to find a regression function.

3.

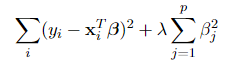
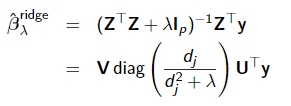
No. In a predictable dataset, a model can be created simply, and would not require regularization. Regularization is used to reduce the randomness, but if the data doesn’t have that, it is not necessary and otherwise complicates the solution.

4.

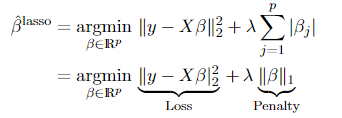
L1 (Lasso) and L2 (ridge) regularization methods.

5.

Ridge Regression minimizes this formula:

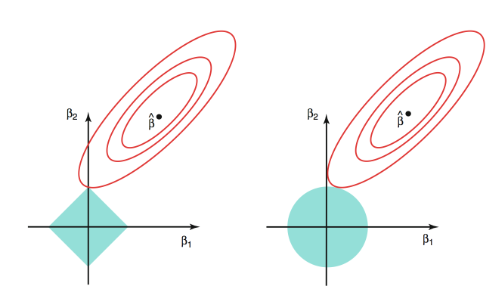
derived solution: 

Lasso Minimizes:



6.

LASSO Ridge



Lasso solutions are sparse, therefore naturally preforming variable selection and regularization. In the above picture, only 1 Beta value is recorded in LASSO. Ridge regression, does not reduce the feature set, but preforms regularization. So, the advantage of lasso is that is that you can reduce your feature set, but since features are being eliminated you may lose some precision.

Problem 5.

1.

source: https://www.ncbi.nlm.nih.gov/pubmed/6484379

Ridge regression has been used to study syphilis data, forestry growth, and medical data. The research being published is studying how to weight the error, to find the best predictive model.

2.

This data set consists of three types of entities: (a) the

specification of an auto in terms of various characteristics, (b)

its assigned insurance risk rating, (c) its normalized losses in use

as compared to other cars. The second rating corresponds to the

degree to which the auto is more risky than its price indicates.

Cars are initially assigned a risk factor symbol associated with its

price. Then, if it is more risky (or less), this symbol is

adjusted by moving it up (or down) the scale. Actuarians call this

process "symboling". A value of +3 indicates that the auto is

risky, -3 that it is probably pretty safe.

The third factor is the relative average loss payment per insured

vehicle year. This value is normalized for all autos within a

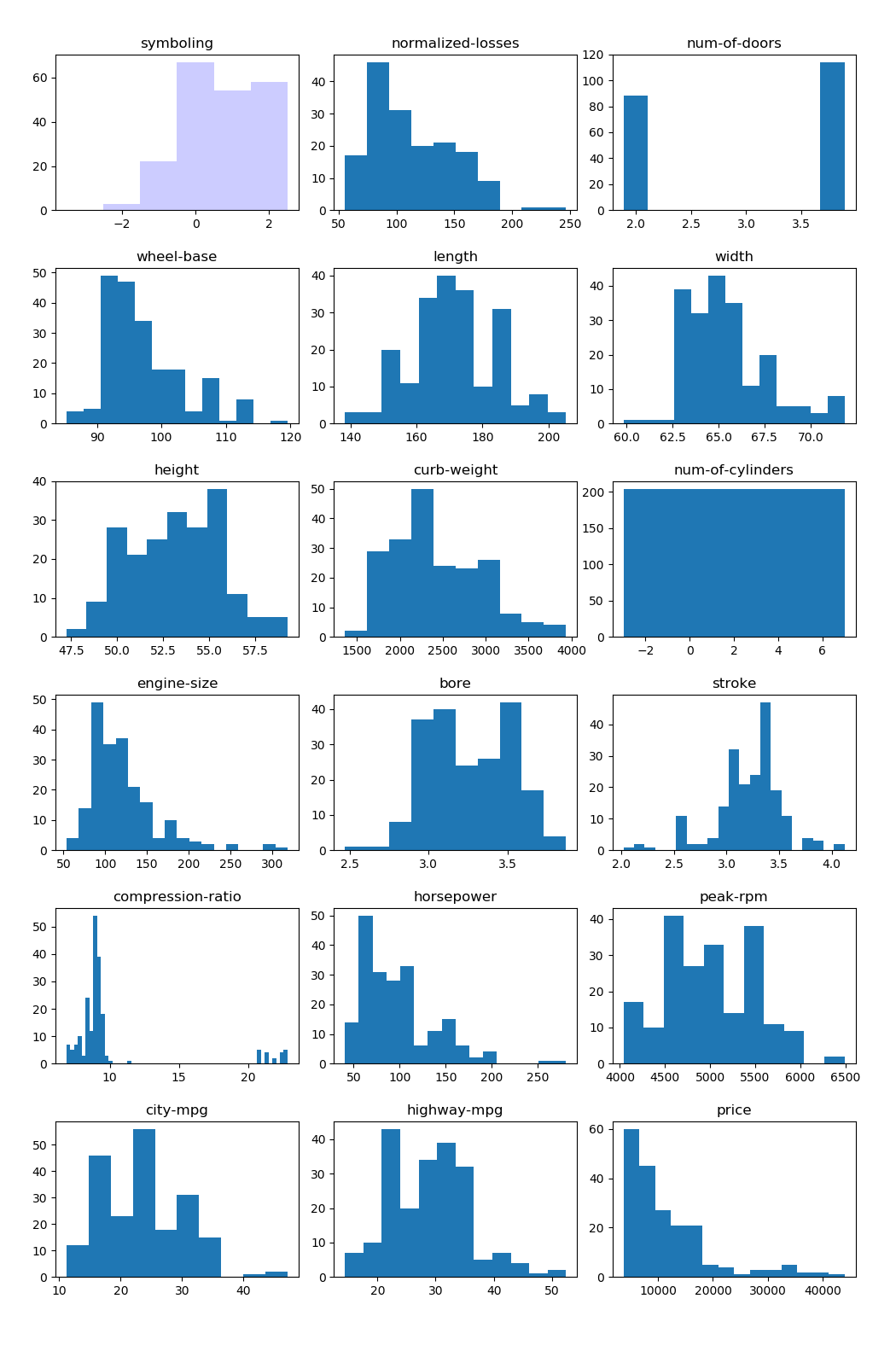
particular size classification (two-door small, station wagons,

sports/speciality, etc...), and represents the average loss per car

per year.

3.

Numerical Value Features: Symboling, num-of-doors, Normalized Losses, wheel-base, length, height, curb-weight, num-of-cylinders, engine-size, bore, stroke, compression-ratio, horsepower, peak-rpm, city-mpg, highway-mpg, price



4.

|  |
| --- |
| def \_solve\_svd(X, y, alpha):   1. U, s, Vt = linalg.svd(X,full\_matrices=False) 2. idx = s > 1e-15 # samedefault value as scipy.linalg.pinv 3. s\_nnz = s[idx][:, np.newaxis] 4. UTy = np.dot(U.T, y) 5. d = np.zeros((s.size, alpha.size), dtype=X.dtype) 6. d[idx] = s\_nnz / (s\_nnz \*\* 2 + alpha) 7. d\_UT\_y = d \* UTy 8. return np.dot(Vt.T, d\_UT\_y).T   Steps 3-8 are just carrying out the minimization solution when Z = UDVT and d values, are the diagonal values of the D (eigenvalue) matrix.  Explanation:   1. Solve the SVD 2. Make sure there are no 0s of the eigenvalues in the coefficient array 3. define the new axis found by the SVD 4. Dot product of the y values to the transpose of the U matrix (right part of solution) 5. Properly size the diagonal array 6. Calculate the middle part of the solution 7. Multiply the middle and right part of the solutions 8. Finish the solution and return the coefficients |

5.

nums\_only = nums\_only.drop("normalized-losses",1) #remove normalized losses because lots of NaN

nums\_only = nums\_only.dropna(0,'any') #remove rows with a NaN

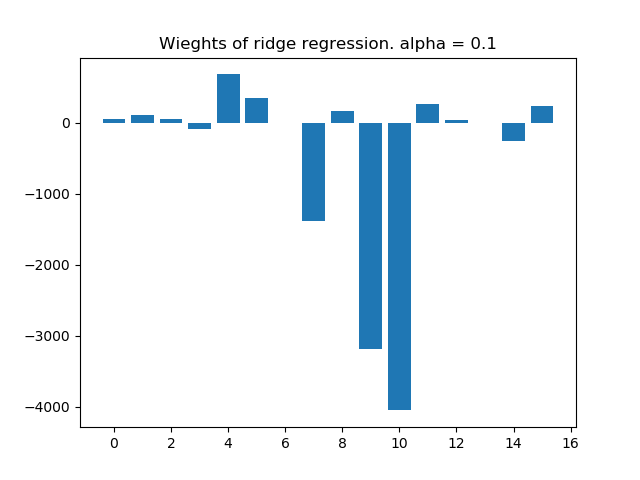
price = nums\_only["price"] #output array

nums\_only = nums\_only.drop("price",1) #input array

reg = lm.Ridge(solver="auto",alpha=0.1) #initialize ridge regression

reg = reg.fit(nums\_only, price) #do the fitting

score = reg.score(nums\_only,price) #extract the scores

6.

0 symboling

1 num-of-doors

2 wheel-base

3 length

4 width

5 height

6 curb-weight

7 num-of-cylinders

8 engine-size

9 bore

10 stroke

11 compression-ratio

12 horsepower

13 peak-rpm

14 city-mpg

15 highway-mpg

0:3 [ 5.12609164e+01, 1.03064932e+02, 5.48768808e+01, -9.08045677e+01]

4:7 [6.81939485e+02, 3.42989003e+02, 9.08940732e-01, -1.37997328e+03]

8:11 [1.69898223e+02, -3.18441498e+03, -4.05442342e+03, 2.65079895e+02]

12:15 [4.24991669e+01, 2.13390544e+00, -2.54745807e+02, 2.30562841e+02]

sum sq error: 0.86

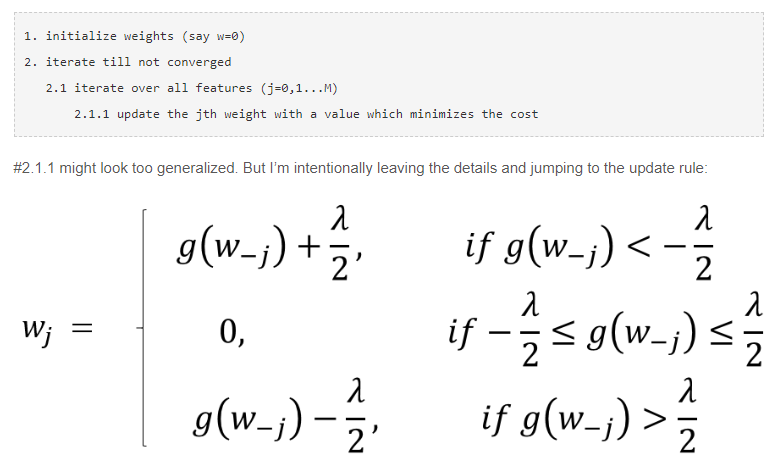
**Three biggest weights** in terms of absolute value: **Stroke, bore, num-of-cylinders**. These all have to do with engine size. These values are strongly negatively correlated, so the smaller they are, the more expensive the car is. The weights are exaggerated, because even though they are very large, the maximum values of stroke, bore, and cylinders are very small. If we normalized the data, there would still be a negative correlation, but the weights would not be so exaggerated because there would not be a need to scale the impact of small number up.

**Smallest weights** absolute value: **wheel-base, curb-height, peak-rpm**. These values individually do not factor into the price of the car.

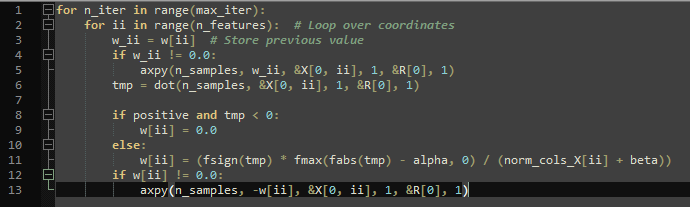
Problem 6.

1.

The figure shows what we are trying to accomplish.



The following code is doing that:



1. Iterate from total iterations
2. Iterate through features
3. Storage weight in temp variable
4. & 5. Update residual array
5. Dot product of residuals to samples
6. Setting weight is dependent of the dot product calculated in 6.
7. Set weight = 0 if no correlation or negativiely correlated
8. Set weight based on the equations in the above figure
9. Update residual array with the new weight

2.

las = lm.Lasso(alpha=10) #initialize lasso regression

las.fit(num\_cp, price\_cp) #do the fitting

las\_score = las.score(num\_cp,price\_cp) #extract the score

print("sum sq error: ",score)

print(las.coef\_)

3.

sum sq err: 0.8541620199936083

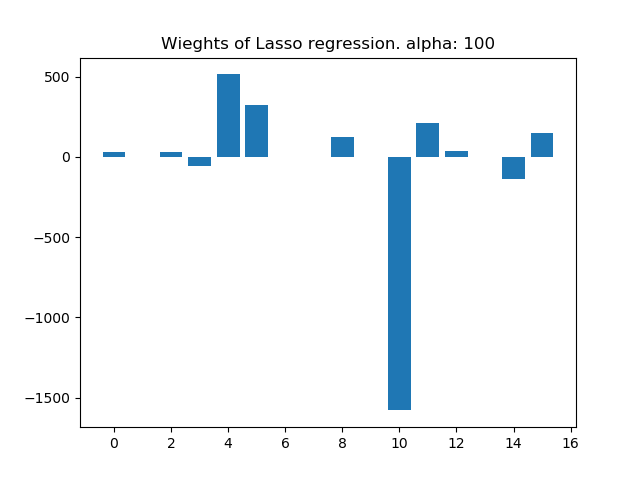
Weights:

[ 32.817 -0. 31.456 -53.232

514.84 323.008 1.652 -0.

123.776 -0. -1580.043 208.975

38.573 2.153 -138.876 151.327 ]

0 symboling

1 num-of-doors

2 wheel-base

3 length

4 width

5 height

6 curb-weight

7 num-of-cylinders

8 engine-size

9 bore

10 stroke

11 compression-ratio

12 horsepower

13 peak-rpm

14 city-mpg

15 highway-mpg

To get the benefits of a simplified model, the alpha was increase to 100. The 3 largest weights are: **stroke, width, height**. The bigger the car, and the smaller the stroke, the more expensive it is going to be, which makes sense.

The 3 smallest weights are 0. **Door number, number of cylinders, bore**

4.

alpha: 0.0001 sum sq err: 0.860105188761832

alpha: 0.001 sum sq err: 0.8601051887586536

alpha: 0.01 sum sq err: 0.8601051884407868

alpha: 0.1 sum sq err: 0.8601051566528157

alpha: 1 sum sq err: 0.8601019776837853

alpha: 10 sum sq err: 0.8597840633848006

alpha: 100 sum sq err: 0.8541620199936083

alpha: 100 sum sq err: 0.8541620199936083

alpha: 1000 sum sq err: 0.8326838917808405

alpha: 10000 sum sq err: 0.8222048646765292

alpha: 100000 sum sq err: 0.7141396215667795

as the alpha increases, the sum sq err does as well. This is because the higher the alpha, the more features get simplified, so the model increases in efficiency, but loses accuracy. The optimal w changes because as alpha gets larger, the sum of the weights absolute value decreases, because the penalty factor has increased in our minimization problem.

Problem 7.

1.

Sources: <https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/>

<https://blog.statsbot.co/support-vector-machines-tutorial-c1618e635e93>

Support vector machine is mostly used for classification, to separate data plotted in a n-D plane. Support Vector machine tries to find the vector of a plane that separates the data the best, by the widest gap possible. This is done by finding the hyperplane so that the distance from the nearest data point on each side is maximized. Outliers can be handled using regression techniques, and the Wikipedia page shows L2 norm being used as the penalty function.

SVM work very well for linear data So if we have non-linear data, we can first transform it so it becomes linear, even if it means going into a higher dimension and run the SVM on that dataset. We can then transform the solution back into the normal space to get a non-linear classifier.

Interestingly, the SVM only needs the dot product of each pair of points to find a classifier. This gives SVM’s the capability to use a kernel function to increase the efficiency when we transform a non-linear set of data points to a higher dimensional space.

2.

#source: http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC

import numpy as np

X = np.array([[-1, -1], [-2, -1], [1, 1], [2, 1]])

y = np.array([1, 1, 2, 2])

from sklearn.svm import SVC

clf = SVC(gamma='auto')

clf.fit(X, y)

SVC(C=1.0, cache\_size=200, class\_weight=None, coef0=0.0,

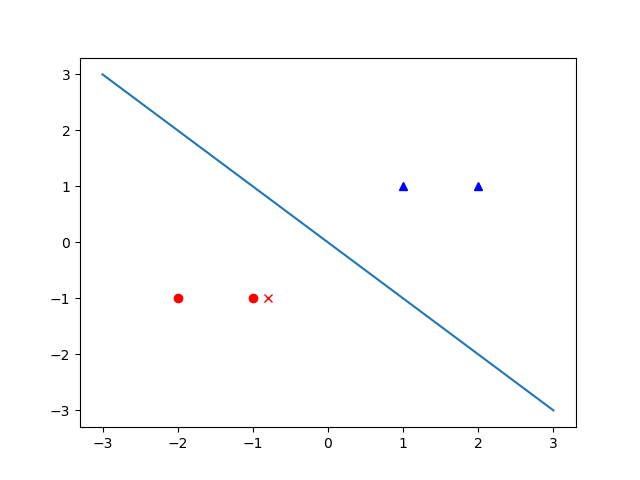
decision\_function\_shape='ovr', degree=3, gamma='auto', kernel='rbf',

max\_iter=-1, probability=False, random\_state=None, shrinking=True,

tol=0.001, verbose=False)

print(clf.predict([[-0.8, -1]]))

The above code is a quick example of an SVM. The X array is the coordinates in a 2D space, and the y array is the class that the points belong too. The X and y array are the training data. [-.8, -1] is the test data, and is classified in the first grouping. The test point is plotted with an ‘x’ in the graph on the next page.

.

Problem 8.

1.

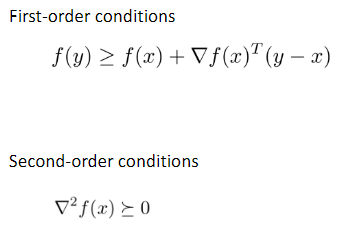
Convex functions ensure that there is only one optimum point and it is the global minimum. These properties are very nice when using an algorithm to find the optimal point, instead of a local (non-global) optimum. Convex functions were applied to radar usage, to determine the shortest distance of British planes for intercepting German bombers.

2.

A function is convex under this property: 

Where 0<θ<1.

We can check if the function is convex with these two other properties:



3.

Yes it is convex.

y’ = 5e5x

y’’ = 25e5x

y’’ is always positive, because exponential functions are always > 0.

4.

f(x) = xln(x) is convex when x>0

5.

Problem 9.

1.

∑

i

(

y

i

−

x

T

i

β

)

2

+

λ

p

∑

j

=1

β

2