Homework 2 – Report

Programming Exercises

Continuation of House Prices from Hw1

1a.

```
# Using OLS
columns_to_include = ['LotFrontage', 'LotArea', 'LotShape_IR2', 'LotShape_IR3', 'LotShape_Reg', 'BldgType_2fmCon', 'BldgType_

target = 'SalePrice'

train_combined = train_combined.drop(columns=['GarageYrBlt'])
train_combined = train_combined.drop(columns=['MasVnrArea'])

#turning the feats into tensors
x_train = np.array(train_combined.drop(columns = [target]), dtype=np.float32)
y_train = np.array(train_combined[target], dtype=np.float32)
x_train = torch.tensor(x_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.float32).reshape(-1,1)

print("NaN values in x_train:", torch.isnan(x_train).any().item())
print("NaN values in y_train:", torch.isnan(y_train).any().item())


class LinearRegression(nn.Module):
    def __init__(self, input_dim):
        super(LinearRegression, self).__init__()
        self.linear = nn.Linear(input_dim, 1)

    def forward(self, x):
        return self.linear(x)

#model instance
input_dim = x_train.shape[1]
model = LinearRegression(input_dim)

#Loss
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.1)
```

```
#train model
num_epochs = 1000
for epoch in range(num_epochs):
    outputs = model(x_train)
    loss = criterion(outputs, y_train)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

with torch.no_grad():
    y_pred = model(x_train).numpy()

#mse
mse = mean_squared_error(y_train.numpy(), y_pred)
root_mse = np.sqrt(mse)
print(f"MSE: {mse}")
print(f"Root MSE: {root_mse}")

#r^2
r2 = r2_score(y_train.numpy(), y_pred)
print(f"r^2 scoreL {r2}")
```

```
NaN values in x_train: False
NaN values in y_train: False
MSE: 2397320192.0
Root MSE: 48962.4375
r^2 scoreL 0.6198825786396085
```

1b. The OLS report did considerably worse than sklearn. The root MSE was 48962 while in sklearn the root MSE was around 23417. Gradient descent is more applicable in linear regression when there's large data sets and more complex loss functions to worry about.

Data generating distribution and convergence of linear regression

2a.

```
#PART A
#made function so it could be used easily in next part

def regression_r2(num_samples):
    #setting up variables
    mean_X = 168
    std_X = 30
    mean_eps = 0
    std_eps = 20
    alpha = 20
    beta = 0.5

    #filling X and epsilon
    X = np.random.normal(mean_X, std_X, num_samples).reshape(-1, 1)
    epsilon = np.random.normal(mean_eps, std_eps, num_samples).reshape(-1, 1)

    #making Y function
    Y = alpha + beta * X + epsilon

    #making prediction
    model = LinearRegression()
    model.fit(X, Y)
    y_predict = model.predict(X)

    #calculating r2
    r2 = r2_score(Y, y_predict)

    return r2, model.coef_[0, 0], model.intercept_[0]
```

To generate synthetic data, I made variables with all the means, std devs., alpha and beta. Then I filled out X and epsilon with the library using the mean and std. dev. since these are not static variables like alpha and beta. Then I made a variable Y, to hold the function. I used a linear regression model to predict the Y values and put that in y_predict. I then made this into a function that returned the r2 and coefficients of the model so I could use it easily in part b.

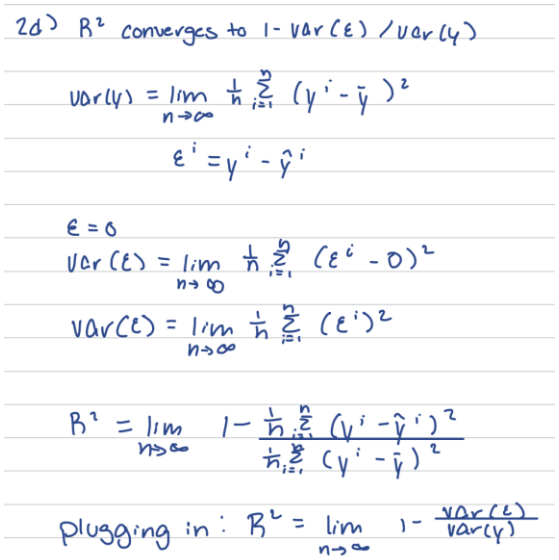
2b.

```
#PART B
print("R2 score, coefficient, intercept:")
print(regression_r2(pow(10, 2)))
print(regression_r2(pow(10, 3)))
print(regression_r2(pow(10, 4)))
print(regression_r2(pow(10, 5)))
print(regression_r2(pow(10, 6)))
```

```
R2 score, coefficient, intercept:
(0.3630184545012368, 0.46843057854924575, 20.989374695179947)
(0.353017675863773 86, 0.483246622048661, 22.081288602609533)
(0.3564574587568712, 0.49569317695165993, 20.5730551643493)
(0.36000505162895113, 0.4999250476535878, 20.070657445872016)
(0.3590619300942529, 0.499414488145987, 20.074551463912044)
```

2c. From part 2b, I observe that beta seems to converge to 0.499 and alpha converges close to 20.07. However, r2 does not converge to 0 but rather 0.35. This means that the model does not fit the data very well.

2d.



2e. Extra Credit

2f. No I don't think a better model exists. The relationship between X and Y is linear (assumed) and adding any more parameters would only add noise rather than making the r2 better.

2g. Some characteristics would be having data that truly fits a linear model, not having a lot of noise, and independence between the variables.

Binary Classification on Text Data

3a. There are 7613 training and 3263 test points. The percentage of training tweets that are real disasters is 42.97% and not real disasters is 57.03%

7612	10873	NaN	

7613 rows × 5 columns

3262	10875	NaN	

3263 rows × 4 columns

```
disaster_count = train[train['target'] == 1].shape[0]
no_disaster_count = train[train['target'] == 0].shape[0]
disaster_percent = (disaster_count/len(train)) * 100
no_disaster_percent = (no_disaster_count/len(train)) * 100
print("disaster percent: ", disaster_percent, "no disaster percent:
```

```
disaster percent:  42.96597924602653 no disaster percent:  57.03402
075397347
```

3b.

```
#PART B
#70% -> 5329/7613 and 30% -> 2284
training_set = train.sample(frac=0.7)
dev_set = train.drop(training_set.index)
print(training_set)
print(dev_set)
```

3c. I have decided to also remove # and hashtags in addition to @ and urls and made a function to clean the data ad run it through the training set and dev set

```
#PART C

def preprocess_data(df):
    words_to_remove = {'the', 'and', 'or'}
    #lowercase
    df['text'] = df['text'].apply(lambda x: x.lower())
    #remove @ and urls
    df['text'] = df['text'].apply(lambda x: re.sub(r'@\S+', '', x))
    #remove # and hashtags
    df['text'] = df['text'].apply(lambda x: re.sub(r'#\S+', '', x))
    #strip punctuation
    df['text'] = df['text'].apply(lambda x: x.translate(str.maketrans('', '', string.punctuation)))
    #strip the and or
    df['text'] = df['text'].apply(lambda x: ' '.join(word for word in x.split() if word not in words_to_remove))
    #Lemmatize
    lemmatizer = nltk.WordNetLemmatizer()
    stop_words = set(stopwords.words('english'))

    def lemmatize_text(text):
        tokens = nltk.word_tokenize(text)
        lemmatized_tokens = [lemmatizer.lemmatize(word) for word in tokens if word not in stop_words]
        return ' '.join(lemmatized_tokens)

    df['text'] = df['text'].apply(lemmatize_text)

    return df


preprocess_data(training_set)
preprocess_data(dev_set)
```

3d.

```
#PART D

M = 2
vectorizer = CountVectorizer(binary=True, min_df=M)

train_vectors = vectorizer.fit_transform(training_set['text'])
dev_vectors = vectorizer.transform(dev_set['text'])

num_features = len(vectorizer.get_feature_names_out())
print(f"Total number of features: {num_features}")
```

```
Total number of features: 4863
```

3e.

i.

```
#PART E

#part i.
lr_nr = LogisticRegression(penalty=None, max_iter=200)
lr_nr.fit(train_vectors, training_set['target'])
y_train_predict_nr = lr_nr.predict(train_vectors)
y_dev_predict_nr = lr_nr.predict(dev_vectors)
f1_train_nr = f1_score(training_set['target'], y_train_predict_nr)
f1_dev_nr = f1_score(dev_set['target'], y_dev_predict_nr)
print('Logistic Regression without regularization:')
print(f"F1 score training: {f1_train_nr}")
print(f"F1 score dev: {f1_dev_nr}")
```

```
Logistic Regression without regularization:
F1 score training: 0.9622266401590458
F1 score dev: 0.6558540293968576
```

I observe overfitting because the training score is high but dev score is low which means that it learned the training very well to the point it cant perform in dev set.

ii.

```
#part ii.
lr_l1 = LogisticRegression(penalty='l1', solver = 'liblinear')
lr_l1.fit(train_vectors, training_set['target'])
y_train_predict_l1 = lr_l1.predict(train_vectors)
y_dev_predict_l1 = lr_l1.predict(dev_vectors)
f1_train_l1 = f1_score(training_set['target'], y_train_predict_l1)
f1_dev_l1 = f1_score(dev_set['target'], y_dev_predict_l1)
print('\nLogistic Regression with L1 regularization:')
print(f"F1 score training: {f1_train_l1}")
print(f"F1 score dev: {f1_dev_l1}")
```

```
Logistic Regression with L1 regularization:
F1 score training: 0.836782690498589
F1 score dev: 0.732824427480916
```

Since both F1 scores are similar, this means the model is performing well in training and developing.

iii.

```
#part iii.
lr_l2 = LogisticRegression(penalty='l2', solver = 'liblinear')
lr_l2.fit(train_vectors, training_set['target'])
y_train_predict_l2 = lr_l2.predict(train_vectors)
y_dev_predict_l2 = lr_l2.predict(dev_vectors)
f1_train_l2 = f1_score(training_set['target'], y_train_predict_l2)
f1_dev_l2 = f1_score(dev_set['target'], y_dev_predict_l2)
print('\nLogistic Regression with L2 regularization:')
print(f"F1 score training: {f1_train_l2}")
print(f"F1 score dev: {f1_dev_l2}")
```

```
Logistic Regression with L2 regularization:
F1 score training: 0.8455089820359282
F1 score dev: 0.7261078483715964
```

Since both F1 scores are similar, this means the model is performing well in training and developing.

iv. The one that performed best on training was the logistic regression model without regularization terms (f1 = 0.96). The one that performed best on the dev set was regression model with L1 regularization (f1 = 0.73). I observed overfitting with the logistic regression without regularization and regularization did help. Before the difference between the f1 scores of the training and dev set was 0.31 and after the difference between the f1 scores once it was regularized was closer to 0.1 difference.

v.

```
#part v
feat_names = vectorizer.get_feature_names_out()
coefficients = lr_l1.coef_[0]

positive = coefficients.argsort()[-5:][::-1]
negative = coefficients.argsort()[:5]

print("\nPositive:")
for words in positive:
    print(f"{feat_names[words]} : {coefficients[words]}")

print("\nNegative:")
for words in negative:
    print(f"{feat_names[words]} : {coefficients[words]}")
```

```
Positive:
typhoon : 4.315161192995197
wildfire : 3.8060819664259316
hiroshima : 3.3046233284259743
spill : 2.959903699664655
migrant : 2.9476537231460322

Negative:
republican : -2.3416731906628385
character : -2.1406858163631273
love : -1.9833625522399099
ebay : -1.9441161683546115
ruin : -1.9390339173765958
```

The most important to determine whether there's a disaster are: typhoon, wildfire, Hiroshima, spill, and migrant

3f.

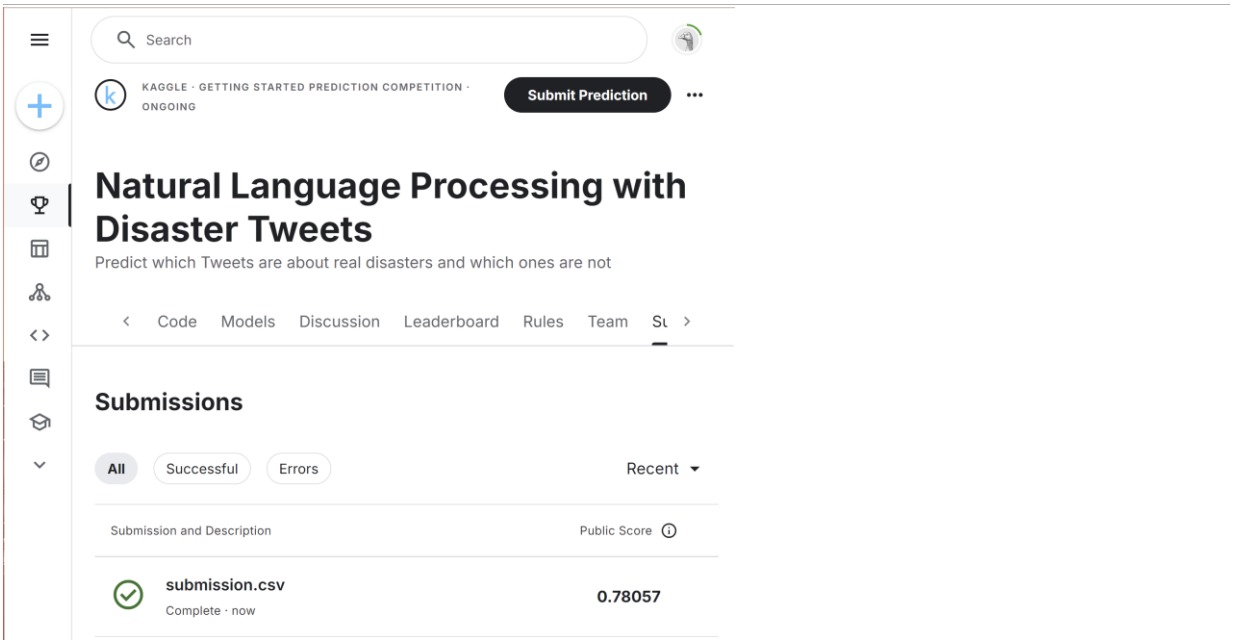
```
#PART F

M = 3
vec_2_gram = CountVectorizer(ngram_range=(2, 2), min_df=M)

#fitting the data
x_train_2_gram = vec_2_gram.fit_transform(training_set['text'])
x_dev_2_gram = vec_2_gram.transform(dev_set['text'])

#length of vocab
vocab_len = len(vec_2_gram.get_feature_names_out())
print(f"Num of 2-grams in vocabulary: {vocab_len}")

#printing the pairs of n-gram
rand_2_grams = np.random.choice(vec_2_gram.get_feature_names_out(), 10, replace=False)
print("\nRandom 2-grams in vocab:")
for thing in rand_2_grams:
    print(thing)

#training the models
lr_2_gram = LogisticRegression(penalty='l2')
lr_2_gram.fit(x_train_2_gram, training_set['target'])

#predicting on training and dev set
y_train_pred_2_gram = lr_2_gram.predict(x_train_2_gram)
y_dev_pred_2_gram = lr_2_gram.predict(x_dev_2_gram)

#f1 scores
f1_train_2_gram = f1_score(training_set['target'], y_train_pred_2_gram)
f1_dev_2_gram = f1_score(dev_set['target'], y_dev_pred_2_gram)

print("\nLogistic regression with 2-gram:")
print(f"F1 score training: {f1_train_2_gram}")
print(f"F1 score dev: {f1_dev_2_gram}")
```

```
Num of 2-grams in vocabulary: 1434

Random 2-grams in vocab:
year fergusons
giant crane
dress meme
california time20150805
vehicle accident
fatal outbreak
reddits new
back life
bar admits
affected fatal

Logistic regression with 2-gram:
F1 score training: 0.6266706266706267
F1 score dev: 0.5426039536468985
```

I chose the threshold M as 3 because it's not a very large dataset and I feel like 2 times could be a coincidence of people writing the same words but a third person verifies whether it is a real accident or not. This model does worse than bag of models because the F1 score of l1 dev, which was the best model was 0.73 and the F1 score of the n-gram dev was 0.54

3g.

```
#PART G

data_comb = pd.concat([training_set, dev_set], ignore_index=True)

M = 3
vectorizer = CountVectorizer(binary=True, min_df=M)
x_complete = vectorizer.fit_transform(data_comb['text'])

# Train the logistic regression model with L1 regularization
lr_l1 = LogisticRegression(penalty='l1', solver='liblinear')
lr_l1.fit(x_complete, data_comb['target'])

# Predict on the train and test sets
x_test = vectorizer.transform(test['text'])
y_pred = lr_l1.predict(x_test)

final = pd.DataFrame({'id': test['id'], 'target': y_pred})
final.to_csv('submission.csv', index=False)
```



The F1 score is a bit higher than I expected, given the previous L1 F1 score, I was expecting something around 0.73

Written Exercises below.

Written Exercises Below

Written Exercises

1) Question 1

Prove:

$$\arg\max_\theta E_{\hat{p}(x,y)}[\log p_\theta(y|x)] = \arg\min_\theta E_{\hat{p}(x)}[KL(\hat{p}(y|x) \| p_\theta(y|x))]$$

Empirical Distribution:

$$\hat{p}(x,y) = \begin{cases} y_n & \text{if } (x,y) \in D \\ 0 & \text{otherwise} \end{cases}$$

Maximizing Likelihood:

$$\arg\max_\theta E_{\hat{p}(x,y)}[\log p_\theta(y|x)]$$

Using Empirical Distribution:

$$E_{\hat{p}(x,y)}[\log p_\theta(y|x)] = \sum_{i=1}^{n} \hat{p}(x^i, y^i) \log p_\theta(y^i|x^i) = \sum_{i=0}^{n} \frac{1}{n} \log p_\theta(y^i|x^i) = \frac{1}{n} \sum_{i=0}^{n} \log p_\theta(y^i|x^i)$$

Model Distribution:

$$KL(\hat{p}(y|x) \| p_\theta(y|x)) = E_{\hat{p}(y|x)}\left[\log \frac{\hat{p}(y|x)}{p_\theta(y|x)}\right]$$

Avg. KL Divergence:

$$E_{\hat{p}(x)}[KL(\hat{p}(y|x) \| p_\theta(y|x))] = E_{\hat{p}(x)}\left[E_{\hat{p}(y|x)}\left[\log \frac{\hat{p}(y|x)}{p_\theta(y|x)}\right]\right]$$

Simplify and Expand:

$$E_{\hat{p}(x)}[KL(\hat{p}(y|x) \| p_\theta(y|x))] = E_{\hat{p}(x)}[\log \hat{p}(y|x) - \log p_\theta(y|x)]$$

Substitute $\hat{p}(y|x)$:

$$E_{\hat{p}(x)}[KL(\hat{p}(y|x) \| p_\theta(y|x))] = E_{\hat{p}(x)}[\log \hat{p}(y|x) - \log p_\theta(y|x)]$$

$$= E_{\hat{p}(x)}\left[\log\left(\frac{1}{n}\sum_{i=1}^{n} \delta(y - y^i)\right) - \log p_\theta(y|x)\right]$$

so when you max $E_{\hat{p}(x,y)}[\log p_\theta(y|x)]$ you reduce the avg. log of

model's prob. and minimizing KL divergence $\mathbb{E}_{\hat{p}(x)}[KL(\hat{p}(y|x) \| p_\theta(y|x))]$ reduces divergence between data distributions

2) Question 2

a. let $\sigma(a) = \frac{1}{1+e^{-a}}$. Show that $\frac{d\sigma(a)}{da} = \sigma(a)(1-\sigma(a))$

$$\frac{d\sigma(a)}{d(a)} = \frac{(1+e^{-a})\cdot 0 - (1)(\frac{d}{da}(1+e^{-a}))}{(1+e^{-a})^2}$$

$$\frac{d}{da}(1+e^{-a}) = 0 + (-e^{-a})(-1) = e^{-a}$$

$$\frac{d\sigma(a)}{d(a)} = \frac{(1+e^{-a})\cdot 0 - (1)(e^{-a})}{(1+e^{-a})^2} = \frac{0 - e^{-a}}{(1+e^{-a})^2} = \frac{-e^{-a}}{(1+e^{-a})^2}$$

rewrite: $1-\sigma(a) = \frac{e^{-a}}{1+e^{-a}}$ and $\sigma(a) = \frac{1}{1+e^{-a}}$

expressing derivative with sigma:

$$\frac{-e^{-a}}{(1+e^{-a})^2} = \frac{-e^{-a}}{(1+e^{-a})} \cdot \frac{1}{(1+e^{-a})} = -\sigma(a)(1-\sigma(a))$$

so: $\frac{\partial\sigma(a)}{da} = \sigma(a)(1-\sigma(a))$

b. derive $\nabla\ell(\theta) = [y - \sigma(\theta^T x)]x$

where $\ell(\theta) = y\log\sigma(\theta^T x) + (1-y)\log(1-\sigma(\theta^T x))$

Using the chain rule:

$$\nabla \ell(\theta) = \frac{d\ell(\theta)}{d\theta} = y(1-\sigma(\theta^T x))x - (1-y)\frac{\sigma(\theta^T x)(1-\sigma(\theta^T x))}{1-\sigma(\theta^T x)}x$$

$$\nabla \ell(\theta) = yx - \sigma(\theta^T x)x \implies \nabla \ell(\theta) = [y - \sigma(\theta^T x)]x$$

3) Question 3

The problem with using single learning rate for all components in the conventional vector update rule for gradient descent is slow convergence because the learning rate will be too small. In higher dimensions, it could diverge instead of converge.

4) Question 4

The conditions under which gradient descent may fail to converge is if the learning rate is too large/slow, gets stuck in saddle points, or is too noisy. The types of convergence behaviors observed depending on step size are: Suitable learning rate → converging to minimum, too large → divergence/oscillation, too small → takes forever

5) Question 5

The rationale behind using learning rate in gradient descent optimization, and the typical decay schedules used is to stop oscillation and avoid overshooting. The schedules are exponential decay, step decay, and inverse time decay. Decaying learning rate impacts convergence by giving stability and escaping local minima.