CS 578 Final Report

Group Member:

Mian Yang(yang1324@purdue.edu)

Shupei Wang(wang3107@purdue.edu)

Xinru Wang(xinruw@purdue.edu)

Raw Dataset Description

Dataset use: http://www.cs.jhu.edu/~mdredze/datasets/sentiment/ We use the electronic products reviews subset.

Dataset Description: The Multi-Domain Sentiment Dataset contains product reviews taken from Amazon.com from electronic products.

Dataset Format: feature: .... feature: #label#:.

Each line is an user review. Each `feature` is a single word and `count` indicates the frequency of the word in this review. Positive and negative reviews are represented by `label == positive` or `label == negative`.

Preprocessing

Process: First we find all unique words(feature) in raw dataset. Then we loop through each observation of the original dataset and record the frequency of each word for the observation.

Output: Two matrix: `X` and `y`.

- `X` contains 552 rows(# of observations) and 12157 columns(# of unique words). So, `X(i,j)` is the frequency of `j`th feature in `i`th observation.
- `y` contains 552 rows which are labels for each observation.

Cross-validation

> *For this preliminary report, we only use bootstrapping*

We first split the dataset to training and test sets:

```
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.2)
```

Then we perform cross-validation on training set.

Bootstrapping

Our bootstrapping algorithm use 5 bootstraps:

```
def bootstrap_score(X_train_val, y_train_val, clf, B=100, metric='accuracy'):
    n = len(y_train_val) # number of training samples
    bs_scores = [] # bootstrap scores
    for round in range(B):
        indices = choices(range(n), k=n) # pick n samples with replacement
        #indices = random.sample(range(n), n)
        X_train = X_train_val[indices]
        y_train = y_train_val[indices]
        X_val = np.delete(X_train_val, list(set(indices)), axis=0)
        y_val = np.delete(y_train_val, list(set(indices)), axis=0)
        train_classifier(X_train, y_train, clf)
        bs_scores.append(test_classifier(X_val, y_val, clf, metric))
    return sum(bs_scores) / B # return the average score
```

k-fold

We use 5-fold for this project:

```
def kfold_score(X_train_val, y_train_val, clf, k=5, metric='accuracy'):
    n = len(y_train_val) # number of training samples
    val_size = math.floor(n / k)
    cv_scores = []
    for round in range(k):
        if round == k - 1:
            indices = range(round * val_size, n)
        else:
            indices = range(round * val_size, (round + 1) * val_size)
        X_val = X_train_val[indices]
        y_val = y_train_val[indices]
        X_train = np.delete(X_train_val, indices, axis=0)
        y_train = np.delete(y_train_val, indices, axis=0)
        train_classifier(X_train, y_train, clf)
        cv_scores.append(test_classifier(X_val, y_val, clf, metric))
    return sum(cv_scores) / k
```

Hyperparameter Tuning

SVM

The SVM algorithm we used is from `scikit-learn`. We tuned the threshold and kernel function during cross validation process. Two kernel functions used for SVM are `linear` and `rbf`. And we use 30 evenly and linearly spaced threshold values in the range of 0.2 to 0.7. So, 60 total different combinations to tune.

We also write another k-fold score method to calculate scores with threshold for SVM.

```
def kfold_withthres(X_train_val, y_train_val, clf, k,threshold):
    n = len(y_train_val) # number of training samples
    val_size = math.floor(n / k)
    cv_scores = []
    for round in range(k):
        if round == k - 1:
            indices = range(round * val_size, n)
        else:
            indices = range(round * val_size, (round + 1) * val_size)
        X_val = X_train_val[indices]
        y_val = y_train_val[indices]
        X_train = np.delete(X_train_val, indices, axis=0)
        y_train = np.delete(y_train_val, indices, axis=0)
        clf.fit(X_train,y_train)
        pred=(clf.predict_proba(X_val)[:,1]>threshold)*1
        cv_scores.append(get_accuracy(y_val,pred))
    return sum(cv_scores) / k
```

During cross validation, we record the accuracies with their corresponding hyperparameters to prepare for plotting.

Logistic Regression

Similar to SVM, we use logistic regression algorithm from `scikit-learn`. Hyperparameters for logistic regression are `penalty` and `C`. Three penalties are used: `l1`, `l2` and `elasticnet`, each with 20 evenly chosen values on a log scale of -3 to -0.5. So, 60 different combinations in total. Like SVM, we record accuracies during cross validation to prepare for plotting.

Neural Network

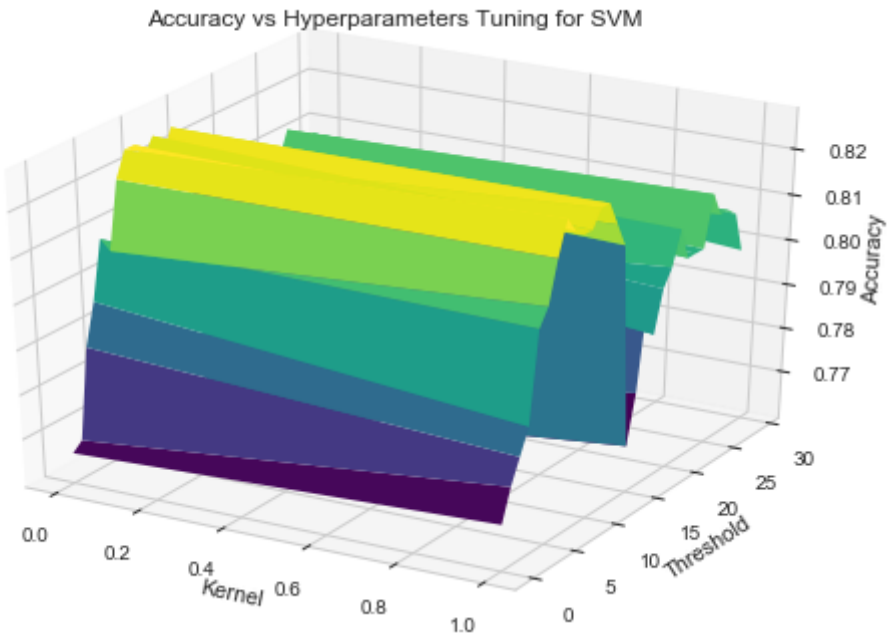
We use a Multi-Layer Perceptron Classifier provided in `scikit-learn` package for this task. Considering the number of samples in our dataset is about 500, we set the solver for MLP to `lbfgs` since it performances better on small dataset and converges faster. Number of hidden layers and number of neurons in each layer are hyperparameters to be tuned. Number of hidden layers are 1, 2, 3 and number of neurons are 10, 50, 100 and 200. So, 12 combinations in total. Like above, accuracies are recorded during cross validation.

Result

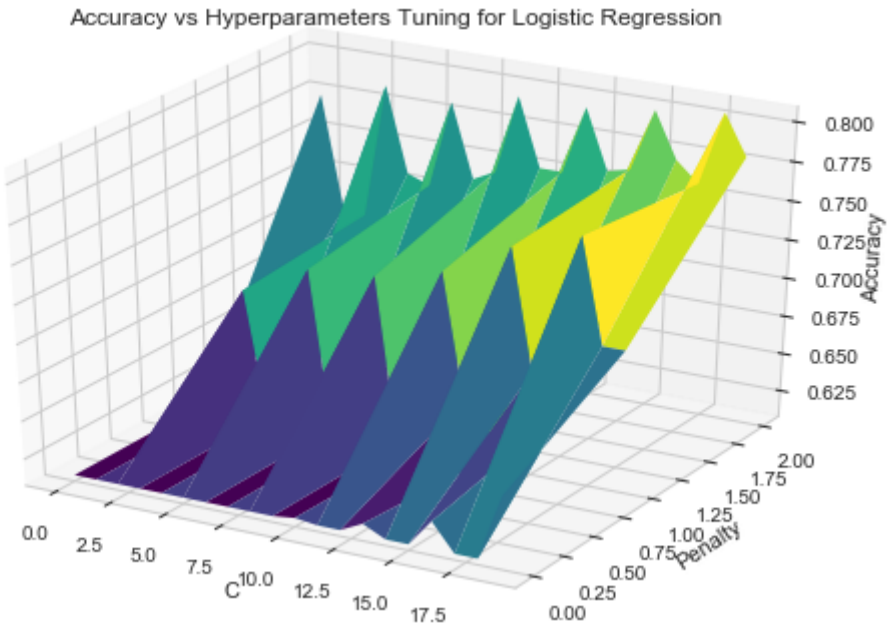
Plots of Hyperparameters vs. Accuracy

By using accuracies collected during cross validation, we show how accuracy is influenced by hyperparameters for each algorithm.

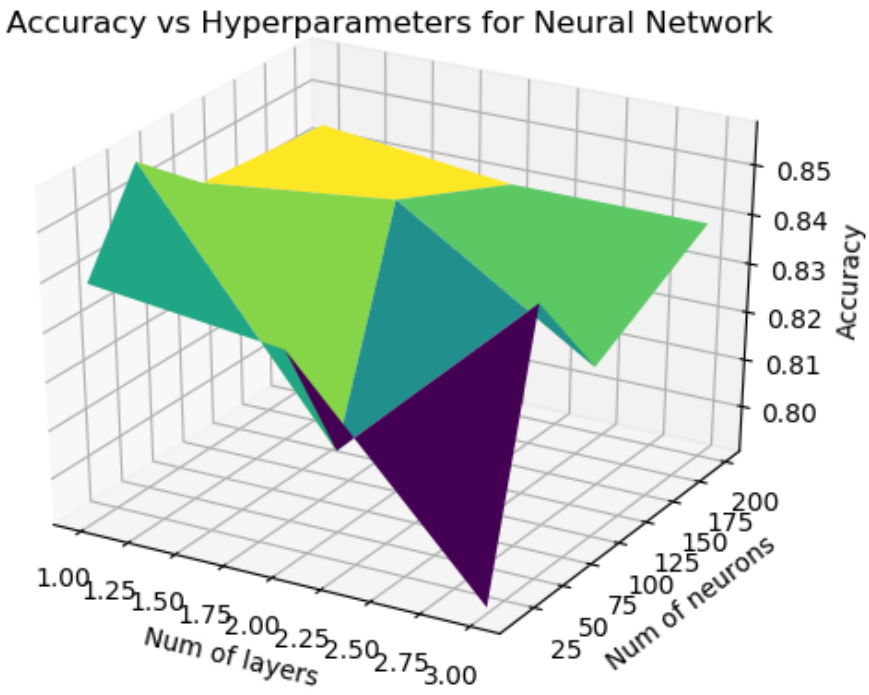
SVM



Logistic Regression



Neural Network

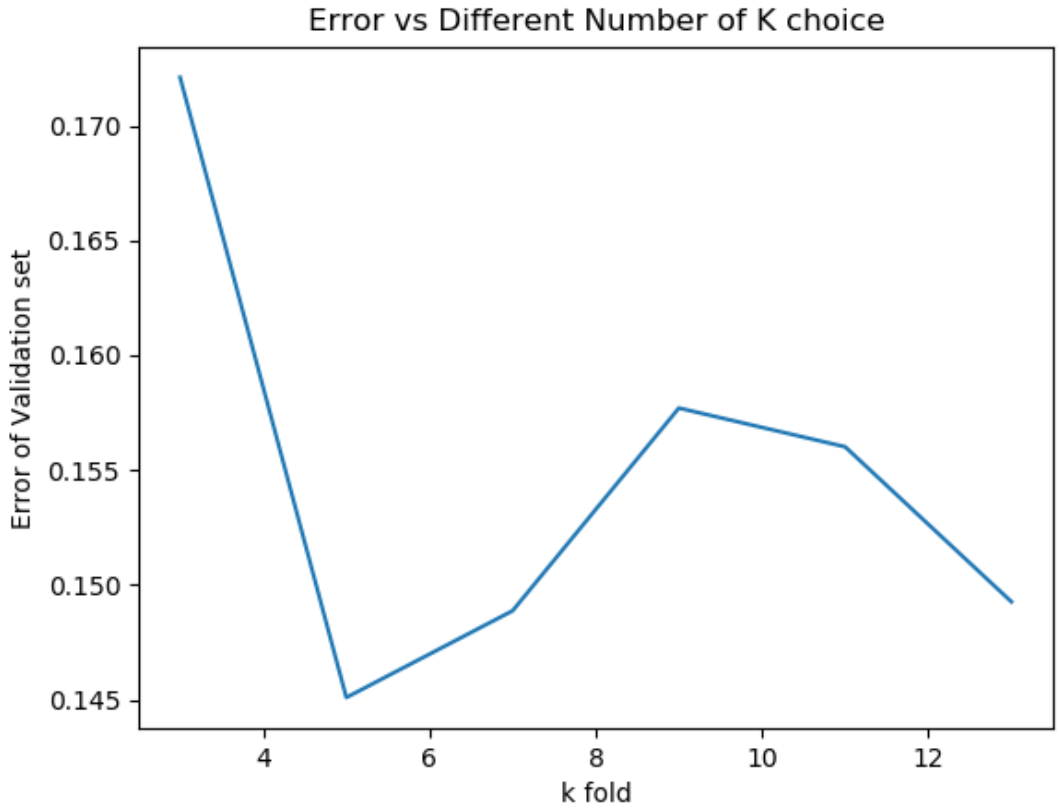


In all these plots, x and y axis are hyperparameters and z axis the the accuracy corresponding to each hyperparameter combination. Overall, all three models have accuracy around 0.8 to 0.85.

> Note that the plot may be different for each run since random samples are selected each time.

Plot of Error vs. Sample Size

To determine whether the error can be influenced by the sample size, we fit samples with different sizes to a SVM classifier with linear kernel. The strategy is reusing the `kfold_score()` and passing different `k` values into it. So, larger `k` value will have smaller sample size.



In this plot, x axis is the value of `k` and y axis is the error of it. We can see the error is changing with the number of samples.

> Note that the plot may be different for each run since random samples are selected each time.

Feature Importance

After training the model, we may want to dive deeper to see why the model makes such predictions. A popular approach to explaining machine predictions is to identify important features. Typically, these explanations assign a value to each feature (usually a word in NLP).

In this project, we adopt two methods for assigning feature importance:

1) built-in feature importance

2) permutation feature importance.

Built-in feature importance is embedded in the machine learning model, such as coefficients in our SVM and logistic regression models.

Permutation feature importance measures the increase in the prediction error of the model after we permuted the feature's values, which breaks the relationship between the feature and the true outcome.

The concept is really straightforward: We measure the importance of a feature by calculating the increase in the model's prediction error after permuting the feature. A feature is "important" if shuffling its values increases the model error, because in this case the model relied on the feature for the prediction. A feature is "unimportant" if shuffling its values leaves the model error unchanged, because in this case the model ignored the feature for the prediction.

The permutation feature importance algorithm based on Fisher, Rudin, and Dominici (2018):

Input: Trained model f, feature matrix X, target vector y, error measure $L(y, f)$.

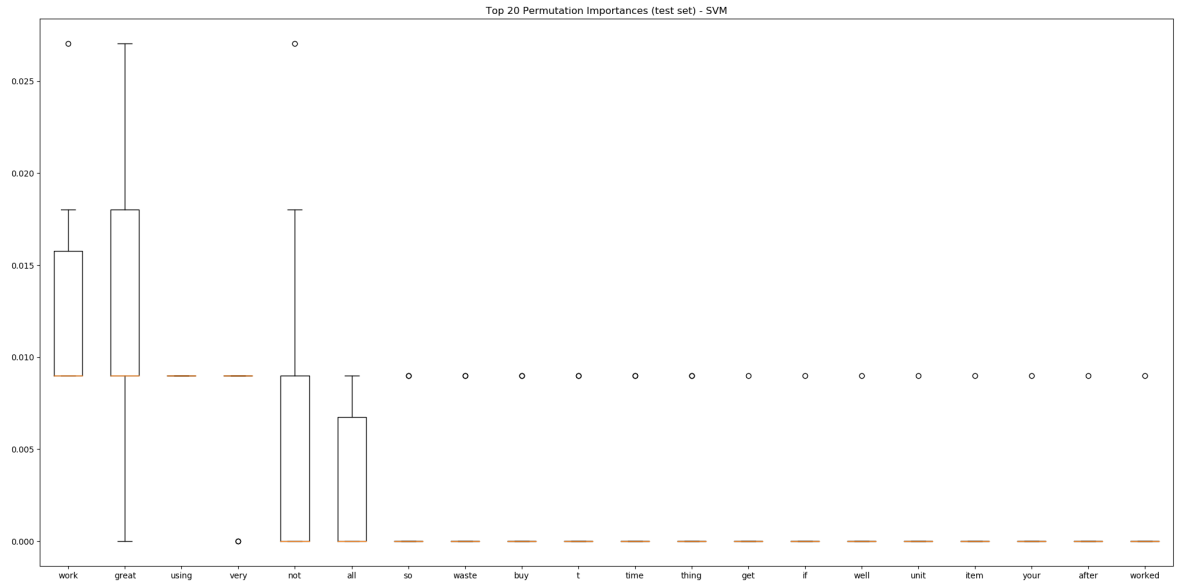
1. Estimate the original model error $e^{orig} = L(y, f(X))$ (e.g. mean squared error)
2. For each feature $j = 1, \ldots, p$ do:
    - Generate feature matrix X^{perm} by permuting feature j in the data X. This breaks the association between feature j and true outcome y.
    - Estimate error $e^{perm} = L(Y, f(X^{perm}))$ based on the predictions of the permuted data.
    - Calculate permutation feature importance $FI_j = e^{perm}/eorig$. Alternatively, the difference can be used: $FI_j = e^{perm} - e^{orig}$
3. Sort features by descending FI.

The following table shows 20 most important features identified by different methods for different models. Note that neural network does not have built-in importance for each feature as SVM and LR.

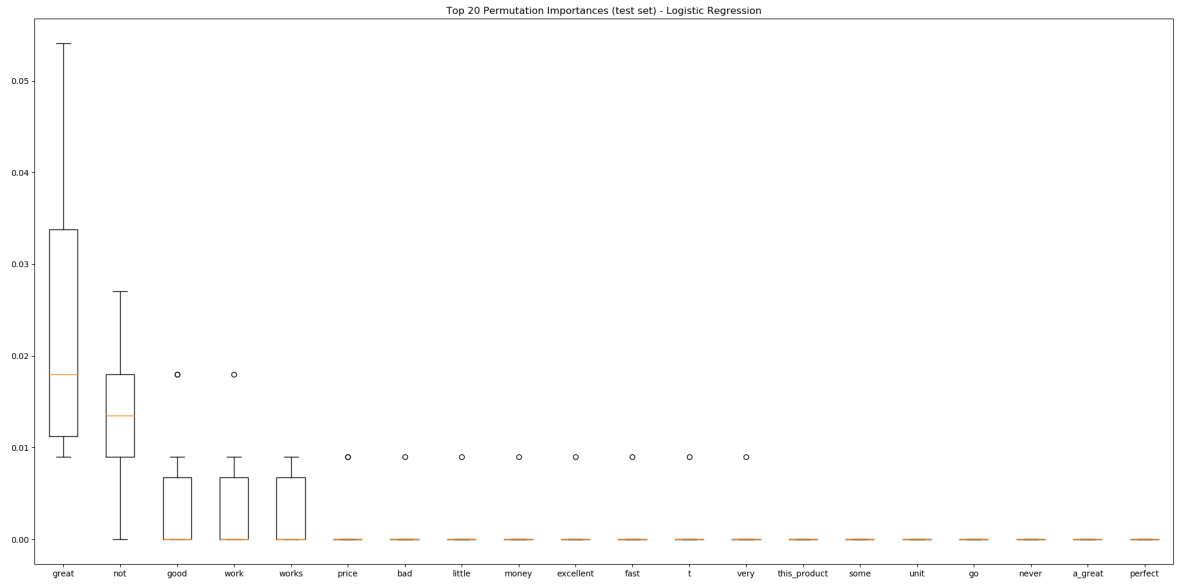
SVM built-in	SVM permutation	NN permutation	LR built-in	LR permutation
great	work	great	great	great
not	great	not	not	not
excellent	using	work	work	good
work	very	very	excellent	work
works	not	using	price	works
t	all	best	t	price
price	so	good	works	bad
good	waste	you	good	little
fine	buy	t	waste	money
garmin	t	waste	money	excellent
money	time	money	out	fast
back	thing	buy	fast	t
fast	get	amazon	fine	very
bad	if	i	love	this_product
value	well	does_not	back	some
junk	unit	just	best	unit
love	item	so	after	go
perfect	your	back	not_work	never
buy_the	after	the_sound	junk	a_great
buy_not	worked	all	very	perfect

We also plot the permutation feature importance scores for each word for each model.

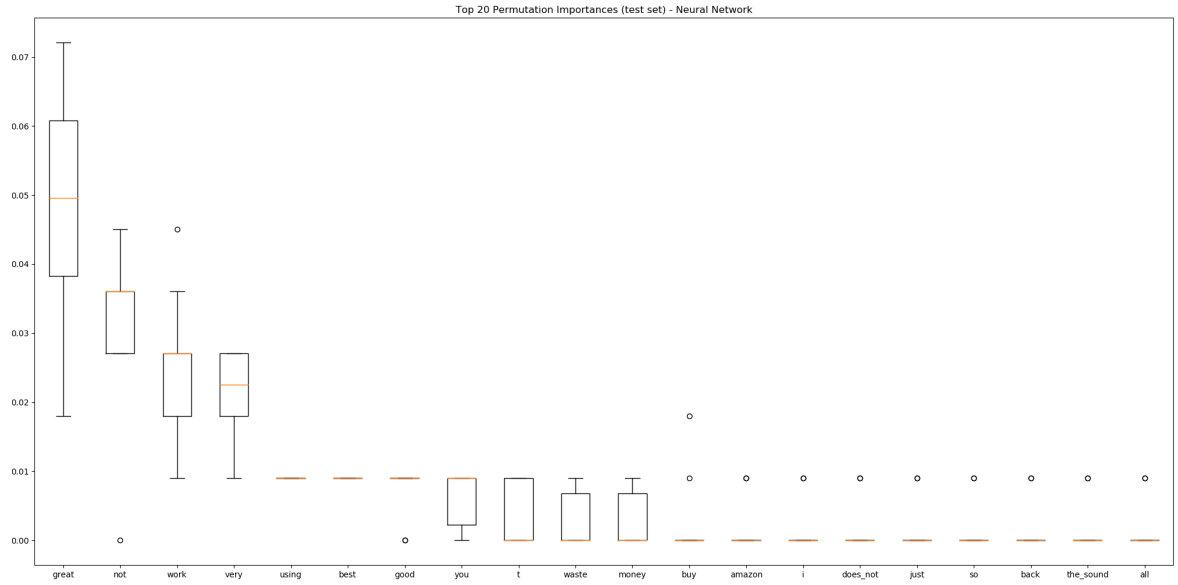
SVM



Logistic Regression



Neural Network



CSV files containing detailed information of words and scores are also generated after running the program.

> Note that the result may be different for each run since random samples are selected each time. Increasing the number of repetition can have more stable results at the cost of longer running time.