CSU44061 Machine Learning - Final Assignment

Samuel Petit - 17333946

Note that the code for all of the approaches in question 1 is provided in the appendix.

1.1 Ignoring languages approach

The first approach I took attempting to tackle this problem was simple in concept. Put simply, to see if it is possible to make predictions on the reviews ignoring the languages of the review text. That is, treat every review the same without taking into account its language.

I then proceeded to load the data from the downloaded dataset into separate X, y and z variables using the code provided with the assignment. For each data point, X is its text review, y is its boolean value representing if it is voted up or not and similarly z is the same for whether a review is early access.

I then proceeded to turn the text reviews into features, to do so I used the CountVectorizer from sklearn as we've seen in class.

```
stem_vectorizer = CountVectorizer(ngram_range=(2, 2))
tokens = stem_vectorizer.fit_transform(X)
```

However I ran into some issues here. To start with, using all of the reviews in all languages resulted in 177,883 individual features per datapoint. This was not scalable nor could I run it on my machine on the entire dataset. I turned to using some stemming such as to reduce the amount of features, this is done by providing an analyser to the vectorizer:

```
stemmer = PorterStemmer()
analyzer = CountVectorizer().build_analyzer()
# Returns the stems from words
def stemmed_words(doc):
    return (stemmer.stem(w) for w in analyzer(doc))

stem_vectorizer = CountVectorizer(analyzer=stemmed_words,
ngram_range=(2, 2))
tokens = stem_vectorizer.fit_transform(X)
```

A point worth noting here is that the stemmer uses the Porter stemming algorithm and may not be optimal for certain languages, more specifically those that use different alphabets. Using stemming I was able to reduce the number of features from 177,883 to 44,853 which is a good improvement. Taking this improvement into account still made my computer struggle to run the models - my first goal was to train a certain number of models such as to identify which ones run the best and identify the best model.

I then turned to taking a sample of the dataset in order to run my experiments, I found that 1000 data samples worked quite well with my machine, it enabled me to obtain relatively fast, meaningful results. I obtain a sample dataset of 1000 data points using the following lines of code:

```
# Dataset has first 5000 reviews positive and next 5000 negative,
shuffle values in unison
X, y, z = shuffle(X, y, z, random_state=0)
# Take a sample 500 reviews such as to avoid too high computing times
when testing approaches
X = X[:1000]
y = y[:1000]
z = z[:1000]
```

Note the fact that I am shuffling the dataset here, that is because I've found that my data was ordered (the first 2500 reviews were positive and the following 2500 negative). I introduced some randomness here by shuffling the data and then taking the first 1000 entries as my sample dataset.

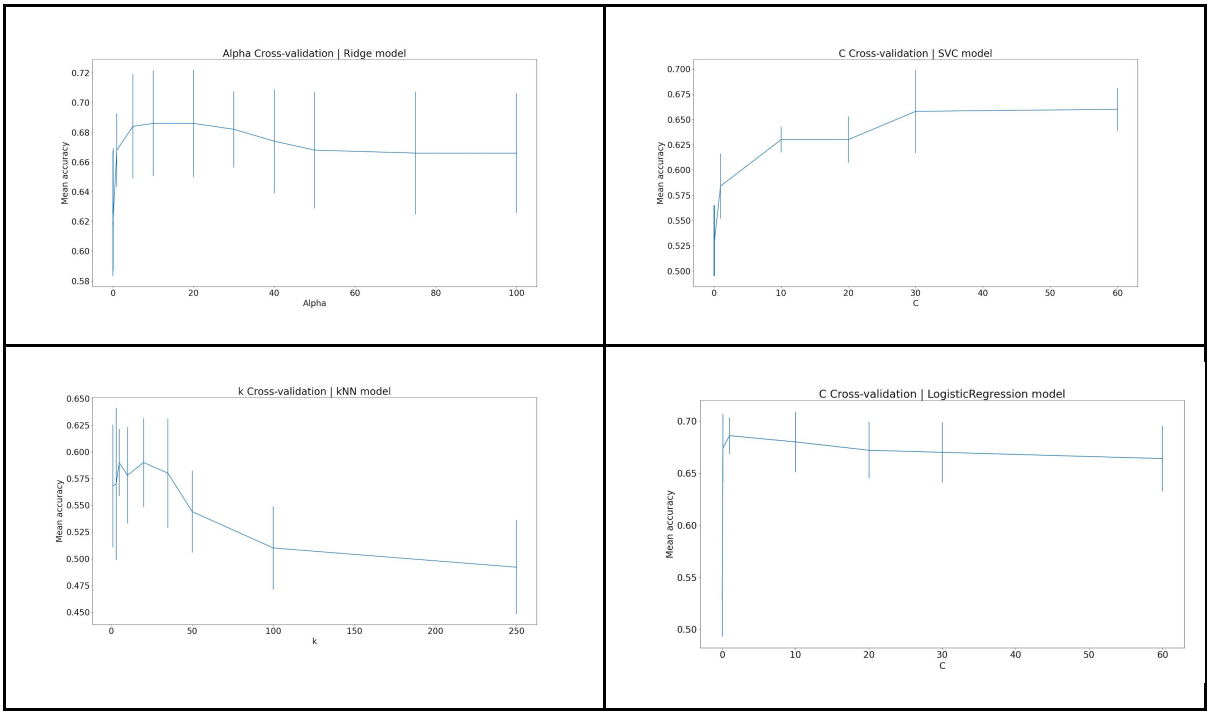
Once I got to this point, I was able to train some models and obtain some plots. I chose to train the following models from sklearn:

- RidgeClassifier
- SVC
- LogisticRegression
- KNeighborsClassifier
- DecisionTreeClassifier
- DummyClassifier

Which are some of the main types of models we've seen in class and should show a wide range of results. I implemented k-fold cross validation in order to train these models. They follow the following system:

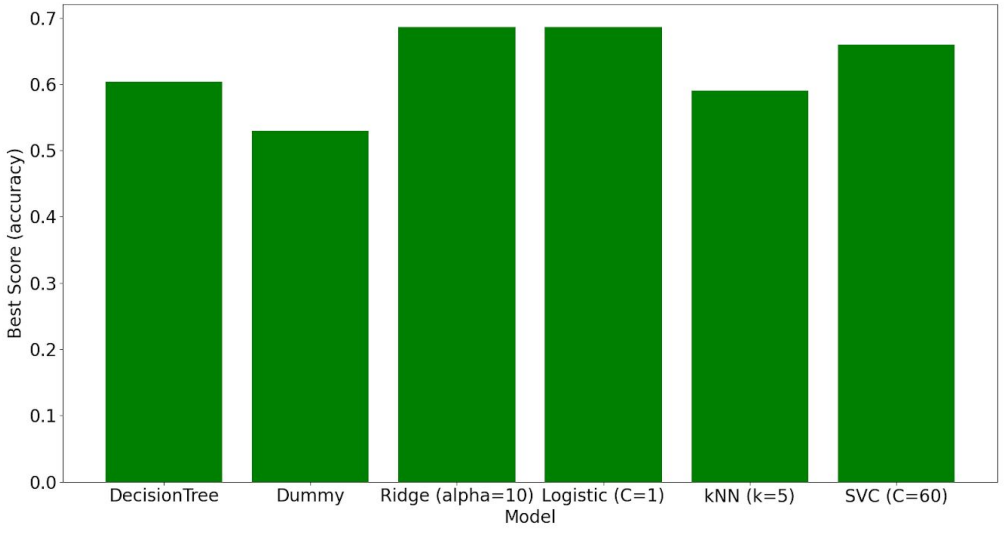
- RidgeClassifier, LogisticRegression, SVC: train models using a range of values for their associated loss parameter (C or alpha). For each model trained, use k-fold with 5 splits, the best model is then the one which has the best average accuracy over the 5 folds.
- DecisionTreeClassifier: Train the model with no parameter (no max_depth parameter is provided), the accuracy for that model is once again the average accuracy of all 5 k-fold splits.
- kNeighborsClassifier: train models using a range of k values. Keep the most accurate model which, once again, is the one which has the highest average accuracy over 5 folds.
- DummyClassifier: The dummy classifier is simply trained without using k-fold, the entire dataset is used during training and its accuracy is then measured on the entire dataset.

For the models which use hyperparameter selection, the cross-validation graphs are generated:



These were only generated for this report, the parameter selection was done automatically as mentioned previously, by selecting the model with the highest accuracy over 5 folds.

From training these 6 models, we can then compare in a plot the best performing ones for each model type, along with their hyperparameters when appropriate.



Once again, the accuracies shown are an average over 5 folds. We notice that the dummy classifier comes last, with slightly over 50% accuracy, it is however not too far from the kNN and Decision tree models which position themselves closer to 60% accuracy. The two models which ended up obtaining the highest accuracy were the Ridge and Logistic regression models, with an accuracy of about 70% which is higher than I expected as we are completely ignoring the languages in this scenario.

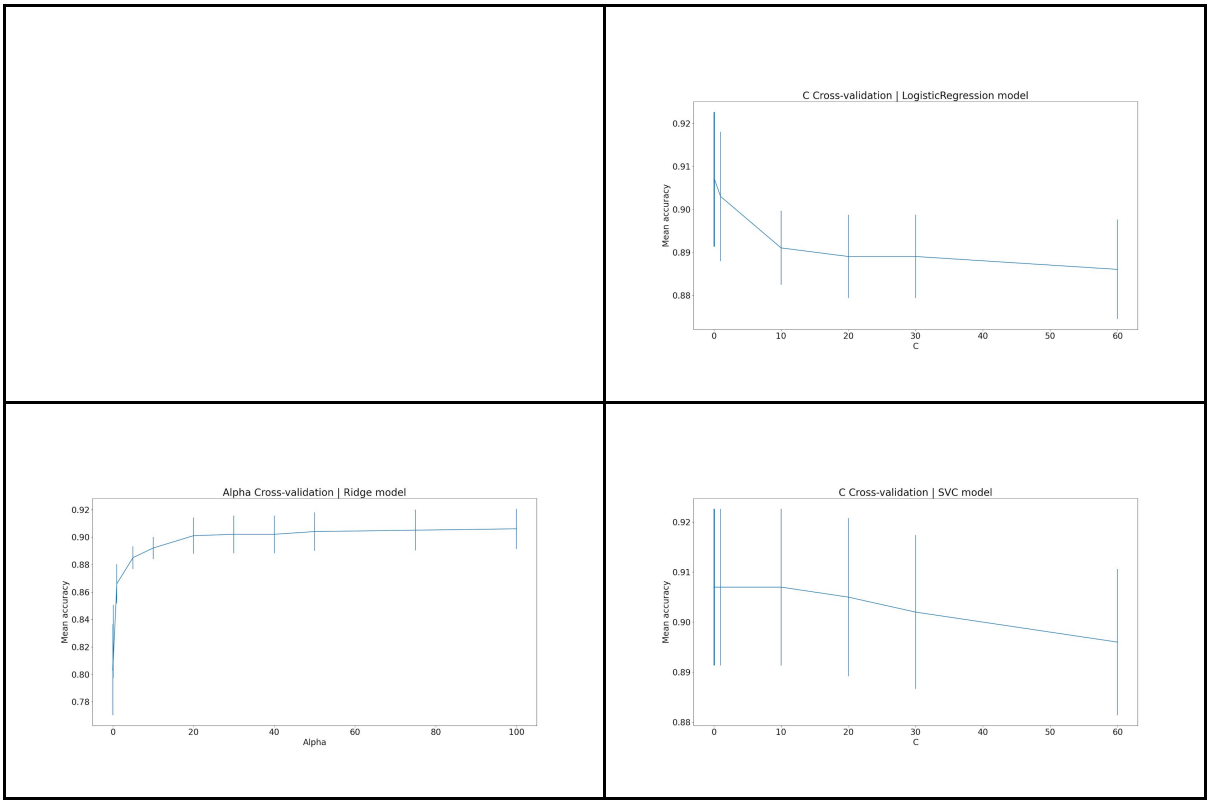
From there I picked the LogisticRegression model (one of the two which performed the highest) with C = 1, and trained it on the entire dataset with the exact same conditions - that is the preprocessing is the same, and 5 splits were used during k-fold. The average accuracy over the entire dataset for the Logistic Regression model using C=1 was of **72%** which is slightly higher than what we had observed during previous training on a sample subset of our data.

Let's now execute this exact same algorithm to attempt to predict the early access metric. The only difference in the code that was used was the following line:

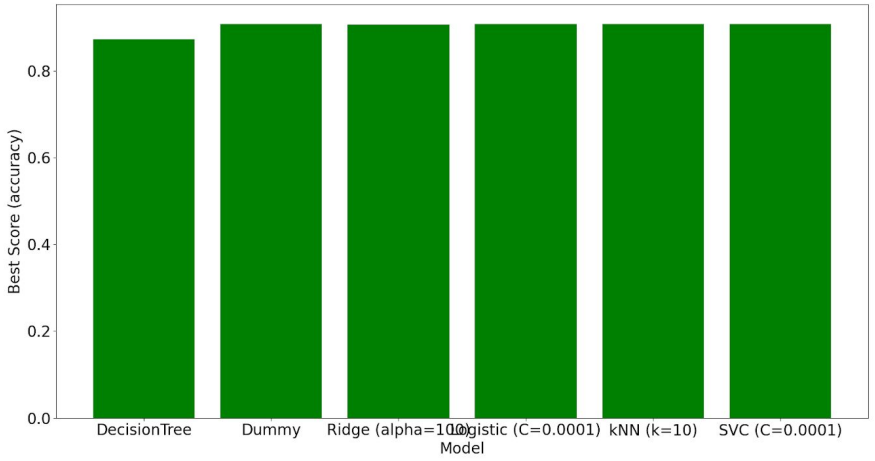
```
y = np.array(y)
# use this line instead of the above one for early access models
# y = np.array(z)
```

Using the line at the bottom instead of the one at the top will make my algorithm attempt to predict early access by using the z metric from the dataset (around line 52 of the ignore_all_languages.py file).

The algorithm is the exact same, so we also obtain cross-validation graphs:



As explained above - parameters are picked automatically using k-fold cross validation with 5 splits. We also obtain the same bar chart which enables us to compare the performance of each of the models:



We notice that the results for early access are much harder to predict, where all models have obtained more or less the same accuracy of about 88% (including the dummy model). And in fact our decision tree model has performed worse than all of the others in this situation.

For this first approach we notice reasonably good performance for predicting whether a review was voted up or not, with most models performing significantly better than the dummy (up to 20% better). However we find that is not the case for predicting early access, while the accuracies seem high at a glance (close to 90%) - we find that this is due to most reviews simply being non-early access, this is confirmed thanks to the performance of the dummy model being the same as the other best performing models.

1.2 Treating each language separately approach

From the first approach, my theory was that the previous models were limited in performance due to the way the processing of text reviews into features was done. Namely - ignoring languages and using a stemming approach that is not language specific.

Hence let's transform my first approach into a system which splits the reviews by language in the hope of obtaining better performing models.
The first step to this approach is finding out what language a specific piece of text is in. To do so, I've decided to use langdetect which enables language detection in about 55 languages, it will return a SO 639-1 language code as its output which enables me to separate each review by language. There is one edge case where some reviews may only contain special characters, or emojis, in this case the detection fails and I label the language as 'other' instead.

We can then start to iterate through all of the languages independently and train models for each of the languages.
There is one other detail worth mentioning with regards to stemming. The SnowballStemmer from nltk (a library we have seen in class) supports a set of languages which also work using algorithms from Martin Porter (similarly to the PorterStemmer we previously used).
The set of supported languages are:

- Danish, Dutch, English, Finnish, French, German, Hungarian, Italian, Norwegian, Portuguese, Romanian, Russian, Spanish and Swedish

Hence, when the current language is supported by the SnowballStemmer, I will set it to use language-specific stemming, in the case that it does not, no stemming is done and instead the tokenization of text will happen on a word by word basis (the default behavior of the CountVectorizer).
The mapping from the language keycode into the language parameter accepted by the SnowballStemmer is fairly straightforward and looks essentially like a switch statement:

```
def get_stemmer_lang(key_lang):
    return {
        'ru': 'russian',
        'en': 'english',
        'de': 'german',
        'pt': 'portuguese',
        'es': 'spanish',
        'fr': 'french',
        'sw': 'swedish',
        'ro': 'romanian',
        'hu': 'hungarian',
        'no': 'norwegian',
        'it': 'italian',
        'fi': 'finnish',
        'da': 'danish',
        'nl': 'dutch',
        'sv': 'swedish'
    }.get(key_lang, None)  # None is default if lang not found
```

The vectorizer is then created using this method's output and feeding it to the SnowballStemmer.

```
stem_lang = get_stemmer_lang(lang)
    if stem_lang is None:
        # Use default analyser if there is no matching stemmer for this language
        analyzer_for_lang = 'word'
    else:
        # Language has a stemmer
        analyzer_for_lang = stemmed_words
        # Redefine stemmer with specified language
        stemmer = SnowballStemmer(stem_lang)
    stem_vectorizer = CountVectorizer(
        analyzer=analyzer_for_lang, ngram_range=(2, 2))
```

Notice how the analyser can become 'lang' instead of the stemming method, this will make tokenization not use stemming when the language is not supported.

Finally, the method for stemming is the exact same as in the first approach:

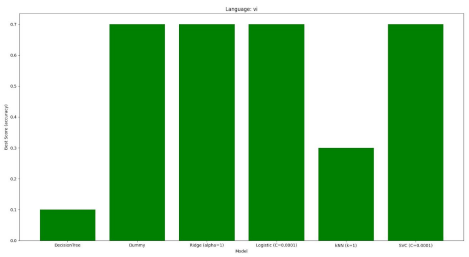
```
stemmer = PorterStemmer()
analyzer = CountVectorizer().build_analyzer()

# Returns the stems from words
def stemmed_words(doc):
   return (stemmer.stem(w) for w in analyzer(doc))
```

We are now ready to map our texts for each specific language into an array of features. For each language I use the exact same code as in the first approach in order to train a set of models for each individual language. For more details, please refer to the above section however in short, these models are trained using k-fold cross validation for selecting hyperparameters with 5 splits (or folds). The models trained for each language are the same as well: RidgeClassifier, SVC, LogisticRegression, KNeighborsClassifier, DecisionTreeClassifier and DummyClassifier. For each language I keep track of the best performing model for each of these and obtain comparison graphs, we also obtain cross validation graphs for every single model which require hyperparameter selection. Once again for more details about all this please refer to this first section above as it is the exact same - the only difference being that it is repeated for every single language.

It is worth noting that for each language the best performing model is kept (the one with the highest average accuracy over 5 folds) such that we can keep track of the most performing model for all of the languages. I also do this for all of the dummy models so that we can compare those performances against the models with the highest accuracy for each language.

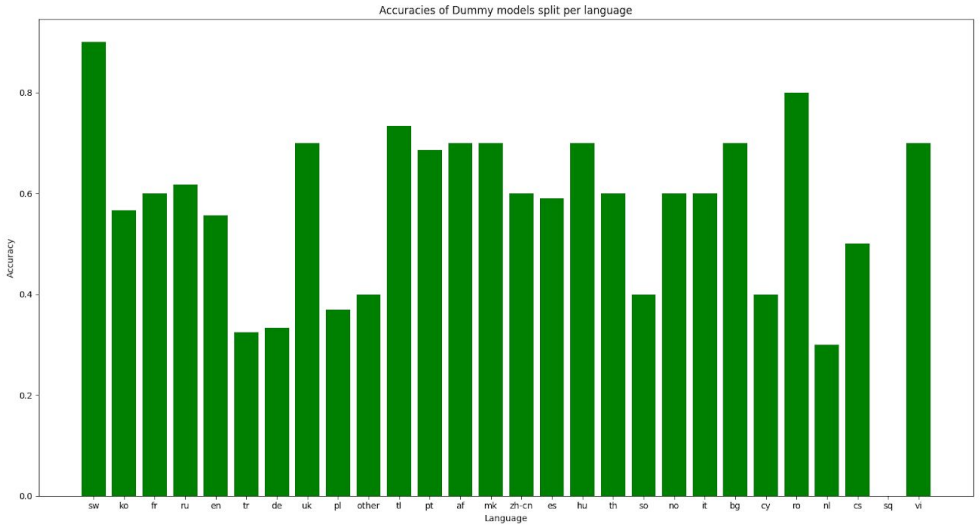
The final detail is that not all languages will include all of the models, when a model fails I will ignore it and move on to the next. For example, this can happen when attempting to train a logistic regression model with a single label in the training output, or when attempting to train a kNN model using a k higher than the amount of available features (in this situation the previous k values are still taken into account). This can happen due to the fact that we are separating per language and sampling, hence some languages may contain very few reviews.

Similarly to the first approach we obtain k-fold cross validation plots, however the hyperparameters are automatically picked thus these were not used for manual selection. To top it off, there are about 4 cross validation plots per language so it would not be ideal to have to go through about 60 plots in order to pick parameters. Similarly, for each language a comparison bar chart is generated an enables comparing the performance of each model type per language, here are 2 samples from the 20 or so that were generated:

We notice quite a big difference in model performances in comparison to what we observed in the first approach, with much more varying accuracies.
The more interesting plot I believe, is this following one which plots each of the languages and the accuracy of its best performing model:

We notice very varying performance differences, going from roughly 30% for Dutch (code nl) all the way close to 90% for languages like Swedish (code sw), Macedonian (code mk) or Czech (code cz). We notice that most of the languages reach about 60% accuracy which is not terrible but also not ideal. Comparing these performances with the dummy models:



We notice a strong spike for the Swedish language which explains the high spike in the previous graph, however we can also see that on average accuracies are much lower, with many being around the 30-40% accuracy range.
Let’s dig into exactly which models obtained the best accuracies for each of the languages and how the dummy model for that language performed. The following table shows details

for each language including the best performing model type, its accuracy and the accuracy for the dummy model. In green we find the languages where trained models perform better than the dummy model, in yellow, we find those where the best model is also the dummy model OR the accuracies are equal.

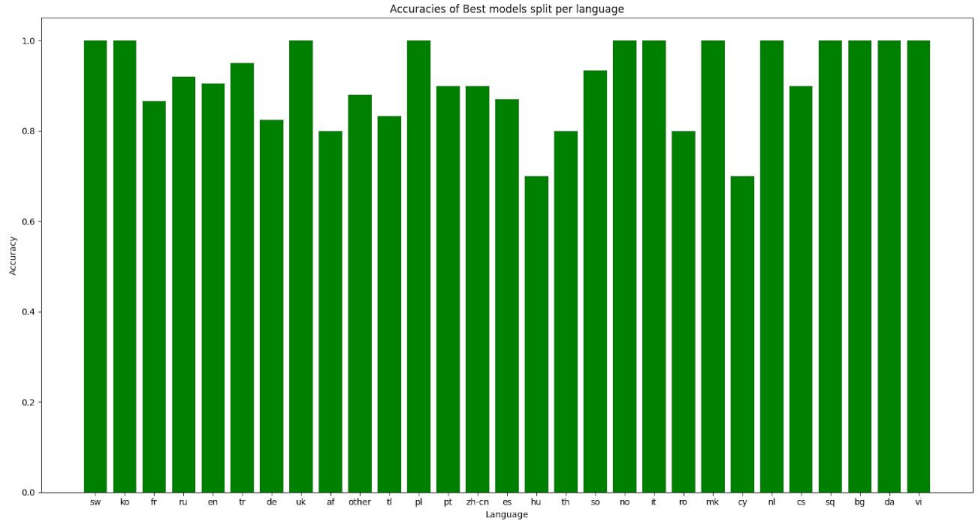
Language code	Best Model	Best Model accuracy	Dummy Model accuracy
sw	DecisionTreeClassifier	0.9	0.9
ko	DummyClassifier	0.567	0.567
fr	DummyClassifier	0.6	0.6
ru	LogisticRegression(C=10)	0.79	0.6175
en	SVC(C=60)	0.747	0.557
tr	KNeighborsClassifier(n_neighbors=3)	0.475	0.325
de	DecisionTreeClassifier	0.6	0.3333
uk	DummyClassifier	0.7	0.7
pl	KNeighborsClassifier	0.52	0.37
other	KNeighborsClassifier(n_neighbors=1)	0.6	0.4
tl	DecisionTreeClassifier	0.733	0.733
pt	KNeighborsClassifier(n_neighbors=5)	0.736	0.686
af	DummyClassifier	0.7	0.7
mk	SVC(C=10)	0.9	0.7
zh-cn	DummyClassifier	0.6	0.6
es	DecisionTreeClassifier	0.76	0.59
hu	DecisionTreeClassifier	0.7	0.7
th	DummyClassifier	0.6	0.6
so	SVC(C=1)	0.8	0.4
no	DecisionTreeClassifier	0.6	0.6
it	DummyClassifier	0.6	0.6
bg	DecisionTreeClassifier	0.7	0.7

cy	DecisionTreeClassifier	0.6	0.4
ro	DummyClassifier	0.8	0.8
nl	DummyClassifier	0.3	0.3
cs	DecisionTreeClassifier	0.9	0.5
sq	KNeighborsClassifier(n_neighbors=3)	0.4	0.0
vi	DummyClassifier	0.7	0.7

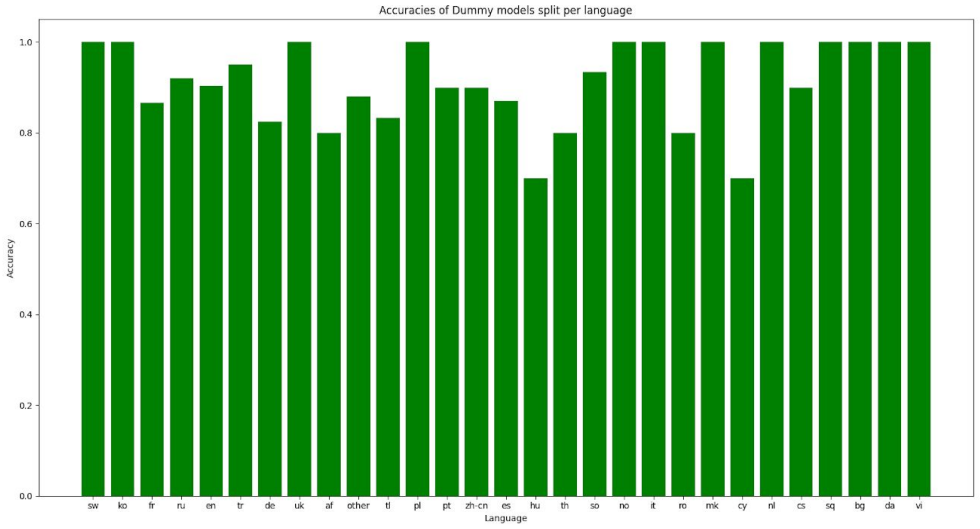
We find that 16 languages do not have models which perform better than the dummy models and only 12 languages have models which outperform the dummy model.

This is quite worrying as it seems splitting by language did not make our predictions better for a lot of languages. We do notice though that certain languages have models which outperform by far the dummy model, languages such as ‘so’ of ‘cs’ for example have 40% or 50% more accuracy than the dummy model.

Once again, repeating this same exact process for attempting to predict early access, we obtain the following results. Plotting one again the best models for each language:



We notice a surprising amount of models get to 100% accuracy, we might note that this may be due to some languages simply not having any reviewers that have early access. Let’s compare with the dummy models performances in order to make sure:



We notice that they look like almost the same plot, meaning that it is very likely that our trained models were outperformed by the dummy models in a lot of languages when attempting to predict the early access metric. Let's look at the comparison table in order to make conclusions about this experiment:

Language code	Best Model	Best Model accuracy	Dummy Model accuracy
sw	DecisionTreeClassifier	1	1
ko	DecisionTreeClassifier	1	1
fr	DummyClassifier	0.867	0.867
ru	DummyClassifier	0.92	0.92
en	SVC(C=20)	0.905	0.903
tr	DecisionTreeClassifier	0.95	0.95
de	DummyClassifier	0.824	0.824
uk	DecisionTreeClassifier	1	1
pl	DecisionTreeClassifier	1	1
other	DecisionTreeClassifier	0.88	0.88
tl	DecisionTreeClassifier	0.833	0.833
pt	DummyClassifier	0.9	0.9
mk	DecisionTreeClassifier	1	1
zh-cn	DecisionTreeClassifier	0.9	0.9

es	DummyClassifier	0.869	0.869
hu	DecisionTreeClassifier	0.7	0.7
th	DecisionTreeClassifier	0.8	0.8
so	DecisionTreeClassifier	0.933	0.933
no	DecisionTreeClassifier	1	1
it	DecisionTreeClassifier	1	1
cy	DecisionTreeClassifier	0.7	0.7
ro	DummyClassifier	0.8	0.8
nl	DecisionTreeClassifier	1	1
cs	DecisionTreeClassifier	0.9	0.9
sq	DecisionTreeClassifier	1	1
vi	DecisionTreeClassifier	1	1
bg	DecisionTreeClassifier	1	1
da	DecisionTreeClassifier	1	1

We notice that the results are noticeably worse than predicting whether a review was voted up, with the dummy models performing as well as the best models for all languages with the exception of english where there is a very small accuracy difference of 0.2% - this could be due to the sampling of data but for each run and increasing the sample size did not seem to make much of a change here.

Conclusion

We can conclude for certain that this approach (and the other) were **not able** to predict whether a user has early access or not based on the review text. This conclusion also goes for the first approach which was not able to produce models with significantly better accuracies than the dummy model with regards to predicting the early access metric.

With regards to predicting the positiveness of a review though, results are much better overall for both approaches. With the first approach, ignoring the language of a review, we find that we were able to train models which outperformed the baseline model by far, achieving an accuracy of 72% over the entire dataset for all languages.

The second approach which attempts to split reviews by their languages has a conflicted output, where models for some languages outperform the dummy models by a large margin, and yet slightly more than half of the languages models were not able to outperform the

baseline model, we can conclude that with regards to this approach, the performance of the models we can obtain will highly depend on the accuracy of the language detector as well as the quality of the preprocessing which may lead to worse results if not accurate, some languages may also have more specific positive and negative words while other languages may be more vague which will affect the accuracy of models.

2.1.

Underfitting a model is essentially training a model where the provided data is not enough for it to make accurate, generalised predictions. That is, the model hasn't learned enough from the data.
An example of this could be attempting to predict future COVID 19 cases only using past counts of cases. This is underfitting as that data is likely not sufficient to make accurate predictions for the future; ideally other parameters such as government regulations, vaccination rates and such should be taken into account.

Overfitting a model is the opposite, it is training a model with too many parameters (too much data) for the problem it is tackling. In this case, the model learns too much from the data and cannot generalise accurately its predictions. In some ways it is tuned to our sample data.
An example of this could be to attempt to make predictions on linear data, using a 50th degree polynomial. This will make our prediction line incredibly close to our training data however it will not behave well on any other data (hence it does not generalise well).

2.2

The following pseudo code executes k-fold cross validation for a sample model. Starting by splitting our data into k parts. Then iterating through folds, forming the datasets to use as training and testing data, training a model for each fold. We finally compute the average accuracy at the end of all folds.

```
let x // features
let y // outputs
dataLen = x.length // our amount of data points
numsFolds = k // amount of folds to use (k) - an integer

// get the size of each fold (floor to keep an integer with no overflow)
kSize = floor(dataLen / numsFolds)
foldsX = new Array(numsFolds) // new array of size folds
foldsY = new Array(numsFolds) // new array of size folds
for i in range(numsFolds) {
    // Let subset(start index, end index) return the subset between the provided indexes.
   foldsX[i] = x.subset(i * kSize, (i + 1) * kSize)
   // note that foldsX and foldsX are arrays of arrays.
   foldsX[i] = y.subset(i * kSize, (i + 1) * kSize)
}
// Copy remaining data points into the last fold
```

```
for (int i = kSize * numsFolds; i < dataLen; i++) {
   foldsX[numsFolds - 1].append(x[i])
   foldsY[numsFolds - 1].append(y[i])
}
// Let's use the accuracy as our measured metric
accuracies = []

// Iterate through the folds
for i in range(numsFolds) {
   // Use the folds to make train / test datasets
   xtrain = []
   xtest = []
   ytrain = []
   ytest = []
   for j in range(numsFolds) {
      // Use the index i as the test data, put the rest into the train data.
      if (j == i) {
         xtest = foldsX[j]
         ytest = foldsY[j]
      } else {
         // Let concat be a function which merges 2 arrays
         xtrain = concat(xtrain, foldsX[j])
         ytrain = concat(ytrain, foldsY[j])
      }
   }

    // Use xtrain and ytrain to train a sample model.
    model = new model().fit(xtrain, train)
    // Use xtest and ytest to test the model. Add the current model accuracy to our set of
   // accuracies.
   accuracies.append(model.accuracy(xtest, ytest))
}
 // Return the average of all the resulting metrics (the accuracy in this case)
// as our final metric value.
averageAccuracy = sum(accuracies) / length(accuracies)
print(averageAccuracy)
```

2.3

Data used to train and test models is noisy. Obtaining multiple outputs for each of the folds enables use to obtain
an average of how a model will perform on different pieces of data, thus getting us a better idea as to how that model will perform.

maybe use this instead:

Cross validation consists of training models using a range of values for its hyperparameters. Using k-fold cross validation means that for each value in the range considered for the hyperparameter in question, we take the average output of all folds as our actual output measure (the model's accuracy for example).
This enables our data for the cross validation to be more accurate thanks to using the average output of k-fold, such as to obtain a set of metrics to select hyperparameters which don't over/underfit our models.

2.4

Reasons why Logistic Regression is better than kNN

- Logistic Regression is faster than kNN to predict outputs, kNN needs to keep track of neighbor points and compute the average of k points, the logistic regression on the other hand only needs to keep track of a set amount of coefficients which only use multiplication and addition to compute its output. This is especially true when k is large.
- kNN models require much more memory to run as it needs to keep track of all neighbors from the training data in order to make its predictions. Logistic regression only needs to keep track of coefficients (a single value per feature).
- Logistic Regression can give a confidence level for its predictions while kNN cannot.
- kNN heavily relies on feature selection. Logistic regression also does but training weights which minimise the loss function will learn to ignore prioritise certain features over time. kNN will simply compute the distance between data points using its associated distance function without weighing the different features.

Reasons why kNN is better than Logistic Regression

- Logistic Regression only works with linearly separable data, kNN models works with non linear data.
- kNN models are easier to train, the only parameters to pick are the value of neighbors to consider k and the distance function to use (typically euclidean distance). Logistic regression models can have more complexity in its parameters to pick such as selecting coefficients (during model training), choosing the loss function and loss parameters...
- kNN models do not learn any model or tune coefficients during training, it will simply compute the average of its k closest neighbors for its predictions and henc does not require training, unlike Logistic regression models which tunes parameters (1 per feature) such that it minimizes its loss function.

2.5

Let's imagine a situation where we are attempting to train a kNN classifier where our dataset mostly consists of true outputs and a minority of false outputs.
If we were to pick a large enough value for k, we could theoretically have our model always output true as there are few false outputs so that the average prediction would always be true.

Let's now imagine another situation where we have a kNN model with k = 1 and our dataset has a small overlap in both classifications. Our kNN model will take the shape of that noise of data and hence possibly predict false outputs, instead of taking the general shape separating the data. Wrong predictions would be given consistently in this conflicting area, when expecting a generalised output from our model we would instead get outputs that reflect our training data that is not desirable.

Appendix

First approach - Ignoring languages

```
from sklearn.utils import shuffle
from nltk.stem import PorterStemmer
from sklearn.model_selection import KFold
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics import confusion_matrix, accuracy_score, mean_squared_error,
precision_recall_fscore_support, roc_auc_score
from sklearn.svm import SVC
from sklearn.linear_model import RidgeClassifier, LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.dummy import DummyClassifier
import json_lines
import numpy as np
import matplotlib.pyplot as plt
plt.rcdefaults()

X = []
y = []
z = []

with open('final_data', 'rb') as f:
   for item in json_lines.reader(f):
       X.append(item['text'])
       y.append(item['voted_up'])
       z.append(item['early_access'])

# Dataset has first 5000 reviews positive and next 5000 negative, shuffle values in
unison
X, y, z = shuffle(X, y, z, random_state=0)

# Take a sample 500 reviews such as to avoid too high computing times when testing
approaches
X = X[:1000]
y = y[:1000]
z = z[:1000]

stemmer = PorterStemmer()
analyzer = CountVectorizer().build_analyzer()


# Returns the stems from words
```

```
def stemmed_words(doc):
    return (stemmer.stem(w) for w in analyzer(doc))


stem_vectorizer = CountVectorizer(analyzer=stemmed_words, ngram_range=(2, 2))
tokens = stem_vectorizer.fit_transform(X)

# Using stems reduces number of words from 177883 to 44853
print(len(stem_vectorizer.get_feature_names()))

X = np.array(tokens.toarray())
y = np.array(y)
# use this line instead of the above one for early access models
# y = np.array(z)

model_names = []
model_accuracies = []

# Decision Tree Model
accuracies = []
kf = KFold(n_splits=5)
for train, test in kf.split(X):
    model = DecisionTreeClassifier().fit(X[train], y[train])
    ypred = model.predict(X[test])
    accuracies.append(accuracy_score(y[test], ypred))
avg_accuracy = sum(accuracies) / len(accuracies)
print("Decision Tree Model average accuracy: ", avg_accuracy)

model_names.append("DecisionTree")
model_accuracies.append(avg_accuracy)

# Dummy Model
accuracies = []
kf = KFold(n_splits=5)
for train, test in kf.split(X):
    model = DummyClassifier(strategy="most_frequent").fit(X[train], y[train])
    ypred = model.predict(X[test])
    accuracies.append(accuracy_score(y[test], ypred))
avg_accuracy = sum(accuracies) / len(accuracies)
print("Dummy Model average accuracy: ", avg_accuracy)

model_names.append("Dummy")
model_accuracies.append(avg_accuracy)

# RidgeClassifier Model
best_ridge_accuracy = (-1, -1)
mean_error = []
std_error = []
alpha_range = [0.0001, 0.01, 0.1, 0, 1, 5, 10, 20, 30, 40, 50, 75, 100]
for alpha in alpha_range:
    accuracies = []
    kf = KFold(n_splits=5)
    for train, test in kf.split(X):
        model = RidgeClassifier(alpha=alpha).fit(X[train], y[train])
        ypred = model.predict(X[test])
        accuracies.append(accuracy_score(y[test], ypred))
```

```
    avg_accuracy = sum(accuracies) / len(accuracies)
    print("Ridge Model, alpha = ", alpha,
          " - average accuracy: ", avg_accuracy)
    mean_error.append(np.array(accuracies).mean())
    std_error.append(np.array(accuracies).std())
    if avg_accuracy > best_ridge_accuracy[0]:
        best_ridge_accuracy = (avg_accuracy, alpha)

# plot the CV
fig = plt.figure()
plt.rc('font', size=20)
ax = fig.add_subplot(111)
plt.errorbar(alpha_range, mean_error, yerr=std_error)
ax.set_ylabel("Mean accuracy")
ax.set_xlabel("Alpha")
ax.set_title("Alpha Cross-validation | Ridge model")

model_names.append("Ridge (alpha=" + str(best_ridge_accuracy[1]) + ")")
model_accuracies.append(best_ridge_accuracy[0])

# Logistic Model
best_logistic_accuracy = (-1, -1)
mean_error = []
std_error = []
C_range = [0.0001, 0.1, 1, 10, 20, 30, 60]
for C in C_range:
    accuracies = []
    kf = KFold(n_splits=5)
    for train, test in kf.split(X):
        model = LogisticRegression(C=C, max_iter=10000).fit(X[train], y[train])
        ypred = model.predict(X[test])
        accuracies.append(accuracy_score(y[test], ypred))
    avg_accuracy = sum(accuracies) / len(accuracies)
    print("LogisticRegression Model, C = ", C,
          " - average accuracy: ", avg_accuracy)
    mean_error.append(np.array(accuracies).mean())
    std_error.append(np.array(accuracies).std())
    if avg_accuracy > best_logistic_accuracy[0]:
        best_logistic_accuracy = (avg_accuracy, C)

# plot the CV
fig = plt.figure()
plt.rc('font', size=20)
ax = fig.add_subplot(111)
plt.errorbar(C_range, mean_error, yerr=std_error)
ax.set_ylabel("Mean accuracy")
ax.set_xlabel("C")
ax.set_title("C Cross-validation | LogisticRegression model")

model_names.append("Logistic (C=" + str(best_logistic_accuracy[1]) + ")")
model_accuracies.append(best_logistic_accuracy[0])

# kNN Model
best_knn_accuracy = (-1, -1)
mean_error = []
std_error = []
```

```
k_range = [1, 3, 5, 10, 20, 35, 50, 100, 250]
for k in k_range:
   accuracies = []
   kf = KFold(n_splits=5)
   for train, test in kf.split(X):
       model = KNeighborsClassifier(n_neighbors=k).fit(X[train], y[train])
       ypred = model.predict(X[test])
       accuracies.append(accuracy_score(y[test], ypred))
   avg_accuracy = sum(accuracies) / len(accuracies)
   mean_error.append(np.array(accuracies).mean())
   std_error.append(np.array(accuracies).std())
   print("kNN Model, k = ", k,
         " - average accuracy: ", avg_accuracy)
   if avg_accuracy > best_knn_accuracy[0]:
       best_knn_accuracy = (avg_accuracy, k)

# plot the CV
fig = plt.figure()
plt.rc('font', size=20)
ax = fig.add_subplot(111)
plt.errorbar(k_range, mean_error, yerr=std_error)
ax.set_ylabel("Mean accuracy")
ax.set_xlabel("k")
ax.set_title("k Cross-validation | kNN model")

model_names.append("kNN (k=" + str(best_knn_accuracy[1]) + ")")
model_accuracies.append(best_knn_accuracy[0])

# SVC Model
best_svc_accuracy = (-1, -1)
mean_error = []
std_error = []
C_range = [0.0001, 0.1, 1, 10, 20, 30, 60]
for C in C_range:
   accuracies = []
   kf = KFold(n_splits=5)
   for train, test in kf.split(X):
       model = SVC(C=C).fit(X[train], y[train])
       ypred = model.predict(X[test])
       accuracies.append(accuracy_score(y[test], ypred))
   avg_accuracy = sum(accuracies) / len(accuracies)
   mean_error.append(np.array(accuracies).mean())
   std_error.append(np.array(accuracies).std())
   print("SVC Model, C = ", C, " - average accuracy: ", avg_accuracy)
   if avg_accuracy > best_svc_accuracy[0]:
       best_svc_accuracy = (avg_accuracy, C)

# plot the CV
fig = plt.figure()
plt.rc('font', size=20)
ax = fig.add_subplot(111)
plt.errorbar(C_range, mean_error, yerr=std_error)
ax.set_ylabel("Mean accuracy")
ax.set_xlabel("C")
ax.set_title("C Cross-validation | SVC model")
```

```
model_names.append("SVC (C=" + str(best_svc_accuracy[1]) + ")")
model_accuracies.append(best_svc_accuracy[0])

# Plot bar chart to compare performance of all different models
fig = plt.figure()
ax = fig.add_subplot(111)
ax.bar(model_names, model_accuracies, color='green')
ax.set_xlabel("Model")
ax.set_ylabel("Best Score (accuracy)")

plt.show()
```

First approach - training the best model on entire dataset

```
from sklearn.utils import shuffle
from nltk.stem import PorterStemmer
from sklearn.model_selection import KFold
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics import confusion_matrix, accuracy_score, mean_squared_error,
precision_recall_fscore_support, roc_auc_score
from sklearn.svm import SVC
from sklearn.linear_model import RidgeClassifier, LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.dummy import DummyClassifier
import json_lines
import numpy as np
import matplotlib.pyplot as plt
plt.rcdefaults()

X = []
y = []
z = []

with open('final_data', 'rb') as f:
   for item in json_lines.reader(f):
       X.append(item['text'])
       y.append(item['voted_up'])
       z.append(item['early_access'])

# Dataset has first 5000 reviews positive and next 5000 negative, shuffle values in
unison
X, y, z = shuffle(X, y, z, random_state=0)

stemmer = PorterStemmer()
analyzer = CountVectorizer().build_analyzer()


# Returns the stems from words
def stemmed_words(doc):
   return (stemmer.stem(w) for w in analyzer(doc))


stem_vectorizer = CountVectorizer(analyzer=stemmed_words, ngram_range=(2, 2))
```

```
tokens = stem_vectorizer.fit_transform(X)

# Using stems reduces number of words from 177883 to 44853
print(len(stem_vectorizer.get_feature_names()))

X = np.array(tokens.toarray())
y = np.array(y)

# Logistic Model
C = 1
accuracies = []
kf = KFold(n_splits=5)
for train, test in kf.split(X):
   print("Starting fold")
   model = LogisticRegression(C=C, max_iter=10000).fit(X[train], y[train])
   ypred = model.predict(X[test])
   accuracies.append(accuracy_score(y[test], ypred))
avg_accuracy = sum(accuracies) / len(accuracies)
print("LogisticRegression Model, C = ", C,
     " - average accuracy: ", avg_accuracy)
mean_error = np.array(accuracies).mean()
std_error = np.array(accuracies).std()
print("mean error: ", mean_error, " - std error: ", std_error)

# AVG Accuracy 0.72
```

Second approach - Taking languages into account

```
import matplotlib.pyplot as plt
import numpy as np
import json_lines
from sklearn.dummy import DummyClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import RidgeClassifier, LogisticRegression
from sklearn.svm import SVC
from sklearn.metrics import confusion_matrix, accuracy_score, mean_squared_error,
precision_recall_fscore_support, roc_auc_score
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import KFold
from nltk.stem import PorterStemmer
from nltk.stem.snowball import SnowballStemmer
from sklearn.utils import shuffle
from langdetect import detect


# Map a detected language from langdetect into a Stemmer language code (only supported
ones)
def get_stemmer_lang(key_lang):
   return {
       'ru': 'russian',
       'en': 'english',
       'de': 'german',
       'pt': 'portuguese',
```

```
        'es': 'spanish',
        'fr': 'french',
        'sw': 'swedish',
        'ro': 'romanian',
        'hu': 'hungarian',
        'no': 'norwegian',
        'it': 'italian',
        'fi': 'finnish',
        'da': 'danish',
        'nl': 'dutch',
        'sv': 'swedish'
    }.get(key_lang, None)  # None is default if lang not found


stemmer = PorterStemmer()
analyzer = CountVectorizer().build_analyzer()


# Returns the stems from words
def stemmed_words(doc):
    return (stemmer.stem(w) for w in analyzer(doc))


X = []
y = []
z = []

# Read data from file
with open('final_data', 'rb') as f:
    for item in json_lines.reader(f):
        X.append(item['text'])
        y.append(item['voted_up'])
        z.append(item['early_access'])

# Dataset has first positive reviews and then negative (ordered), shuffle values in
unison
X, y, z = shuffle(X, y, z, random_state=0)

# Use a sample of the data for faster tests
X = X[:1000]
y = y[:1000]
z = z[:1000]

# Split reviews into language groups
reviews_by_language = {}
for index in range(len(X)):
    item = X[index]
    try:
        lang = detect(item)
    except:
        # When language not detectable, use "other"
        lang = 'other'
    if lang not in reviews_by_language:
        reviews_by_language[lang] = {
            "x": [],
            "y": [],
```

```
            "z": []
        }
    reviews_by_language[lang]["x"].append(X[index])
    reviews_by_language[lang]["y"].append(y[index])
    reviews_by_language[lang]["z"].append(z[index])

# Values are now stored in reviews_by_language - clear some memory space
X = []
y = []
z = []

best_models = []
dummy_models = []
model = None  # Place holder
# Process each language indepedently
for lang in reviews_by_language.keys():
    print("PROCESSING ", lang)
    stem_lang = get_stemmer_lang(lang)
    if stem_lang is None:
        # Use default analyser if there is no matching stemmer for this language
        analyzer_for_lang = 'word'
    else:
        # Language has a stemmer
        analyzer_for_lang = stemmed_words
        # Redefine stemmer with specified language
        stemmer = SnowballStemmer(stem_lang)
    stem_vectorizer = CountVectorizer(
        analyzer=analyzer_for_lang, ngram_range=(2, 2))
    try:
        tokens = stem_vectorizer.fit_transform(reviews_by_language[lang]["x"])
    except:
        # On tokeniser error, skip the language
        continue
    X = np.array(tokens.toarray())
    y = np.array(reviews_by_language[lang]["y"])
    # use this line instead of the above one for early access models
    # y = np.array(reviews_by_language[lang]["z"])

    # Skip languages with less than 5 reviews (not possible with k-fold)
    # this may trigger depending on sampling size used
    if len(X) < 5:
        continue

    print("training models now...")
    model_names = []
    model_accuracies = []

    # Keep track of the best performing model / accuracy for the current language
    curr_best = (None, -1, 'undefined language')
    # Decision Tree Model
    accuracies = []
    kf = KFold(n_splits=5)
    for train, test in kf.split(X):
        model = DecisionTreeClassifier().fit(X[train], y[train])
        ypred = model.predict(X[test])
        accuracies.append(accuracy_score(y[test], ypred))
```

```
avg_accuracy = sum(accuracies) / len(accuracies)
print("Decision Tree Model average accuracy: ", avg_accuracy)

if curr_best[0] is None or curr_best[1] < avg_accuracy:
    curr_best = (model, avg_accuracy, lang)

model_names.append("DecisionTree")
model_accuracies.append(avg_accuracy)

# Dummy Model
accuracies = []
kf = KFold(n_splits=5)
for train, test in kf.split(X):
    model = DummyClassifier(
        strategy="most_frequent").fit(X[train], y[train])
    ypred = model.predict(X[test])
    accuracies.append(accuracy_score(y[test], ypred))
avg_accuracy = sum(accuracies) / len(accuracies)
dummy_models.append((avg_accuracy, lang))
print("Dummy Model average accuracy: ", avg_accuracy)

if curr_best[0] is None or curr_best[1] < avg_accuracy:
    curr_best = (model, avg_accuracy, lang)

model_names.append("Dummy")
model_accuracies.append(avg_accuracy)

# RidgeClassifier Model
best_ridge_accuracy = (-1, -1)
mean_error = []
std_error = []
alpha_range = [0.0001, 0.01, 0.1, 0, 1, 5, 10, 20, 30, 40, 50, 75, 100]
for alpha in alpha_range:
    accuracies = []
    kf = KFold(n_splits=5)
    for train, test in kf.split(X):
        model = RidgeClassifier(alpha=alpha).fit(X[train], y[train])
        ypred = model.predict(X[test])
        accuracies.append(accuracy_score(y[test], ypred))
    avg_accuracy = sum(accuracies) / len(accuracies)
    print("Ridge Model, alpha = ", alpha,
          " - average accuracy: ", avg_accuracy)
    if curr_best[0] is None or curr_best[1] < avg_accuracy:
        curr_best = (model, avg_accuracy, lang)
    mean_error.append(np.array(accuracies).mean())
    std_error.append(np.array(accuracies).std())
    if avg_accuracy > best_ridge_accuracy[0]:
        best_ridge_accuracy = (avg_accuracy, alpha)

# plot the CV
fig = plt.figure()
plt.rc('font', size=20)
ax = fig.add_subplot(111)
plt.errorbar(alpha_range, mean_error, yerr=std_error)
ax.set_ylabel("Mean accuracy")
ax.set_xlabel("Alpha")
```

```
    ax.set_title("Alpha Cross-validation | Ridge model")

    model_names.append("Ridge (alpha=" + str(best_ridge_accuracy[1]) + ")")
    model_accuracies.append(best_ridge_accuracy[0])

    # Logistic Model
    best_logistic_accuracy = (-1, -1)
    mean_error = []
    std_error = []
    C_range = [0.0001, 0.1, 1, 10, 20, 30, 60]
    for C in C_range:
        accuracies = []
        kf = KFold(n_splits=5)
        try:
            for train, test in kf.split(X):
                model = LogisticRegression(
                    C=C, max_iter=10000).fit(X[train], y[train])
                ypred = model.predict(X[test])
                accuracies.append(accuracy_score(y[test], ypred))
        except:
            # Logistic regression fails when there is a single class (ex all are true) in
the sample data
            break
        avg_accuracy = sum(accuracies) / len(accuracies)
        if curr_best[0] is None or curr_best[1] < avg_accuracy:
            curr_best = (model, avg_accuracy, lang)
        print("LogisticRegression Model, C = ", C,
              " - average accuracy: ", avg_accuracy)
        mean_error.append(np.array(accuracies).mean())
        std_error.append(np.array(accuracies).std())
        if avg_accuracy > best_logistic_accuracy[0]:
            best_logistic_accuracy = (avg_accuracy, C)

    if len(mean_error) > 0 and len(std_error) > 0:
        # plot the CV
        fig = plt.figure()
        plt.rc('font', size=20)
        ax = fig.add_subplot(111)
        plt.errorbar(C_range, mean_error, yerr=std_error)
        ax.set_ylabel("Mean accuracy")
        ax.set_xlabel("C")
        ax.set_title("C Cross-validation | LogisticRegression model")

        model_names.append(
            "Logistic (C=" + str(best_logistic_accuracy[1]) + ")")
        model_accuracies.append(best_logistic_accuracy[0])

        # kNN Model
    best_knn_accuracy = (-1, -1)
    mean_error = []
    std_error = []
    k_range = [1, 3, 5, 10, 20, 35, 50, 100, 250]
    actual_k_range = []
    for k in k_range:
        accuracies = []
        kf = KFold(n_splits=5)
```

```
        try:
            for train, test in kf.split(X):
                model = KNeighborsClassifier(
                    n_neighbors=k).fit(X[train], y[train])
                ypred = model.predict(X[test])
                accuracies.append(accuracy_score(y[test], ypred))
            actual_k_range.append(k)
        except:
            # We need n_neighborhs <= n_samples. Stop kNN execution on exception raised
here
            break
        avg_accuracy = sum(accuracies) / len(accuracies)
        if curr_best[0] is None or curr_best[1] < avg_accuracy:
            curr_best = (model, avg_accuracy, lang)
        mean_error.append(np.array(accuracies).mean())
        std_error.append(np.array(accuracies).std())
        print("kNN Model, k = ", k,
              " - average accuracy: ", avg_accuracy)
        if avg_accuracy > best_knn_accuracy[0]:
            best_knn_accuracy = (avg_accuracy, k)

    # plot the CV
    fig = plt.figure()
    plt.rc('font', size=20)
    ax = fig.add_subplot(111)
    plt.errorbar(actual_k_range, mean_error, yerr=std_error)
    ax.set_ylabel("Mean accuracy")
    ax.set_xlabel("k")
    ax.set_title("k Cross-validation | kNN model")

    model_names.append("kNN (k=" + str(best_knn_accuracy[1]) + ")")
    model_accuracies.append(best_knn_accuracy[0])

    # SVC Model
    best_svc_accuracy = (-1, -1)
    mean_error = []
    std_error = []
    C_range = [0.0001, 0.1, 1, 10, 20, 30, 60]
    for C in C_range:
        accuracies = []
        kf = KFold(n_splits=5)
        try:
            for train, test in kf.split(X):
                model = SVC(C=C).fit(X[train], y[train])
                ypred = model.predict(X[test])
                accuracies.append(accuracy_score(y[test], ypred))
        except:
            # SVC regression fails when there is a single class (ex all are true) in the
sample data
            break
        avg_accuracy = sum(accuracies) / len(accuracies)
        if curr_best[0] is None or curr_best[1] < avg_accuracy:
            curr_best = (model, avg_accuracy, lang)
        mean_error.append(np.array(accuracies).mean())
        std_error.append(np.array(accuracies).std())
        print("SVC Model, C = ", C, " - average accuracy: ", avg_accuracy)
```

```
        if avg_accuracy > best_svc_accuracy[0]:
            best_svc_accuracy = (avg_accuracy, C)

    if len(mean_error) > 0 and len(std_error) > 0:
        # plot the CV
        fig = plt.figure()
        plt.rc('font', size=20)
        ax = fig.add_subplot(111)
        plt.errorbar(C_range, mean_error, yerr=std_error)
        ax.set_ylabel("Mean accuracy")
        ax.set_xlabel("C")
        ax.set_title("C Cross-validation | SVC model")

        model_names.append("SVC (C=" + str(best_svc_accuracy[1]) + ")")
        model_accuracies.append(best_svc_accuracy[0])

    # Plot comparison bar chart of all models for current language
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.bar(model_names, model_accuracies, color='green')
    ax.set_xlabel("Model")
    ax.set_ylabel("Best Score (accuracy)")
    ax.set_title("Language: " + str(lang))

    best_models.append(curr_best)
    print("BEST MODEL WAS: ", curr_best, " -- lang: ", lang)


dummy_accuracy, dummy_langs = zip(*dummy_models)

# Plot performance of dummy model for all languages
fig = plt.figure()
ax = fig.add_subplot(111)
ax.bar(dummy_langs, dummy_accuracy, color='green')
ax.set_xlabel("Language")
ax.set_ylabel("Accuracy")
ax.set_title("Accuracies of Dummy models split per language")

# Plot performance of best models for all languages
models, accuracies, langs = zip(*best_models)
fig = plt.figure()
ax = fig.add_subplot(111)
ax.bar(langs, accuracies, color='green')
ax.set_xlabel("Language")
ax.set_ylabel("Accuracy")
ax.set_title("Accuracies of Best models split per language")

print(best_models)
print(dummy_models)
plt.show()
```