DAT340 - Assignment 4

May 7, 2022

Group PA 47 - Author: Stefano Ribes, ribes@chalmers.se

This Notebook can be viewed online at this link: https://colab.research.google.com/drive/1tGfajmrxsvp4NqQyzJD

1 Programming Assignment 4: Implementing linear classifiers

1.1 Setup

[1]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

[2]:
```
import os

ASSIGNMENT_ID = 'assignment_4'

data_dir = os.path.join(os.path.abspath(''), 'drive', 'MyDrive')
data_dir = os.path.join(data_dir, 'Colab Notebooks', 'dat340', ASSIGNMENT_ID)
data_dir = os.path.join(data_dir, 'data')
if os.path.exists(data_dir):
    print(f'Directory "{data_dir}" exists')
else:
    print(f'WARNING! Directory "{data_dir}" does not exist!')
```

```
Directory "/content/drive/MyDrive/Colab Notebooks/dat340/assignment_4/data"
exists
```

[ ]:
```
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('pdf', 'svg')
```

1.2 Exercise question

Do you have an idea what's going on? Why could the classifier "memorize" the training data in the first case, but not in the second case?

Let’s reproduce the example first.

[ ]:
```python
from sklearn.feature_extraction import DictVectorizer
from sklearn.linear_model import Perceptron
from sklearn.svm import LinearSVC
from sklearn.metrics import accuracy_score
from sklearn.pipeline import make_pipeline

X1 = [{'city':'Gothenburg', 'month':'July'},
      {'city':'Gothenburg', 'month':'December'},
      {'city':'Paris', 'month':'July'},
      {'city':'Paris', 'month':'December'}]
Y1 = ['rain', 'rain', 'sun', 'rain']

X2 = [{'city':'Sydney', 'month':'July'},
      {'city':'Sydney', 'month':'December'},
      {'city':'Paris', 'month':'July'},
      {'city':'Paris', 'month':'December'}]
Y2 = ['rain', 'sun', 'sun', 'rain']

classifier1 = make_pipeline(DictVectorizer(), Perceptron(max_iter=10))
classifier1.fit(X1, Y1)
guesses1 = classifier1.predict(X1)
print(f'Accuracy Perceptron dataset n.1: {accuracy_score(Y1, guesses1)}')

classifier2 = make_pipeline(DictVectorizer(), Perceptron(max_iter=10))
classifier2.fit(X2, Y2)
guesses2 = classifier2.predict(X2)
print(f'Accuracy Perceptron dataset n.2: {accuracy_score(Y2, guesses2)}')

classifier3 = make_pipeline(DictVectorizer(), LinearSVC())
classifier3.fit(X2, Y2)
guesses3 = classifier3.predict(X2)
print(f'Accuracy LinearSVC dataset n.2:  {accuracy_score(Y2, guesses3)}')
```

```
Accuracy Perceptron dataset n.1: 1.0
Accuracy Perceptron dataset n.2: 0.5
Accuracy LinearSVC dataset n.2:  0.5
```

The problem is that the first dataset is **linearly seperable**, whereas the second one is not. Because of that, the linear classifiers aren’t able to correctly separate the points in the second dataset.

We can indeed obtain the same accuracy scores by replacing their labels with numerical values. This also allows us to easily visualize the datapoints in the two datasets.

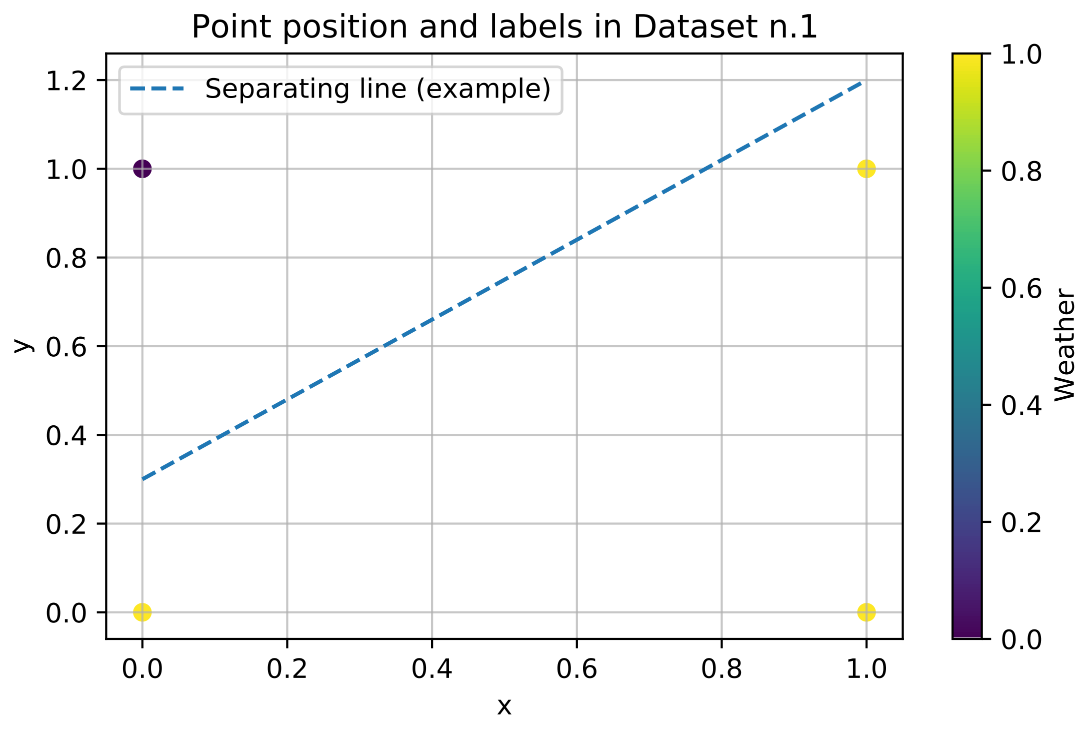
[ ]:
```python
cities = {
    'Paris': 0,
    'Gothenburg': 1,
    'Sydney': 2,
}
months = {
    'July': 1,
    'December': 0,
}
weather = {
    'rain': 1,
    'sun': 0,
}

X1 = [{'city':cities['Gothenburg'], 'month':months['July']},
      {'city':cities['Gothenburg'], 'month':months['December']},
      {'city':cities['Paris'], 'month':months['July']},
      {'city':cities['Paris'], 'month':months['December']}]
Y1 = [weather['rain'], weather['rain'], weather['sun'], weather['rain']]

X2 = [{'city':cities['Sydney'], 'month':months['July']},
      {'city':cities['Sydney'], 'month':months['December']},
      {'city':cities['Paris'], 'month':months['July']},
      {'city':cities['Paris'], 'month':months['December']}]
Y2 = [weather['rain'], weather['sun'], weather['sun'], weather['rain']]

classifier1 = make_pipeline(DictVectorizer(), Perceptron(max_iter=10))
classifier1.fit(X1, Y1)
guesses1 = classifier1.predict(X1)
print(f'Accuracy Perceptron dataset n.1: {accuracy_score(Y1, guesses1)}')

classifier2 = make_pipeline(DictVectorizer(), Perceptron(max_iter=10))
classifier2.fit(X2, Y2)
guesses2 = classifier2.predict(X2)
print(f'Accuracy Perceptron dataset n.2: {accuracy_score(Y2, guesses2)}')

classifier3 = make_pipeline(DictVectorizer(), LinearSVC())
classifier3.fit(X2, Y2)
guesses3 = classifier3.predict(X2)
print(f'Accuracy LinearSVC dataset n.2:  {accuracy_score(Y2, guesses3)}')
```

```
Accuracy Perceptron dataset n.1: 1.0
Accuracy Perceptron dataset n.2: 0.5
Accuracy LinearSVC dataset n.2:  0.5
```

As expected, the scores are the same. Let's now see the points on a 2D plane.

```
[ ]: import matplotlib.pyplot as plt
     import numpy as np

     x_coords = []
     y_coords = []
```

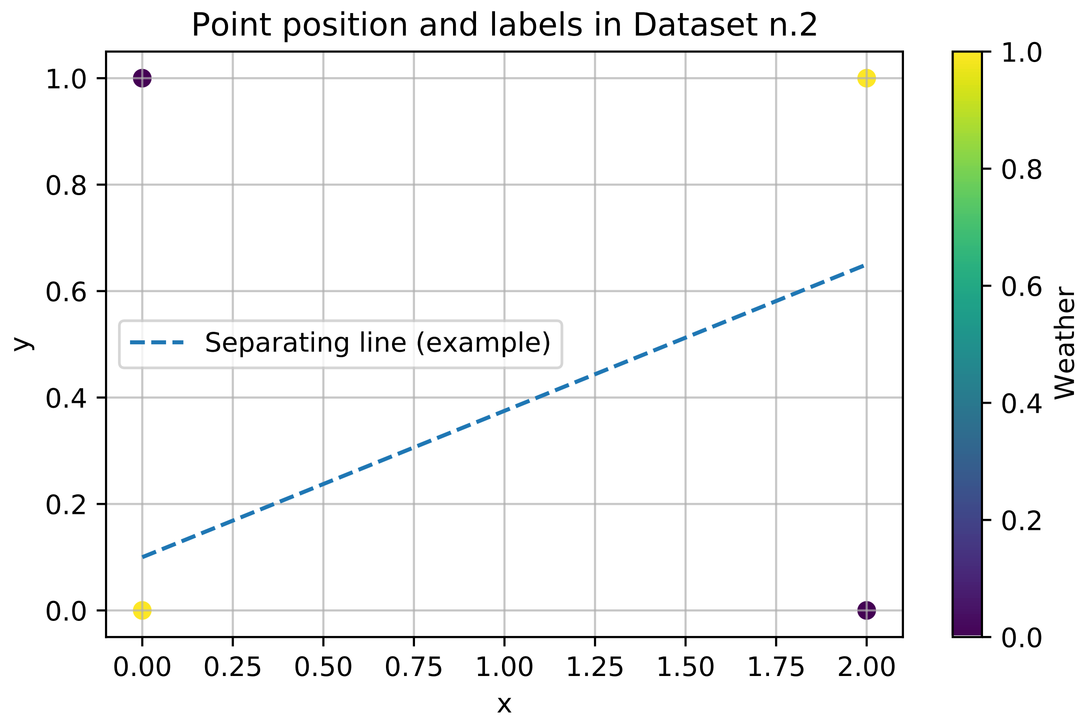
```
for x1 in X1:
    x_coords.append(x1['city'])
    y_coords.append(x1['month'])
plt.scatter(x_coords, y_coords, c=Y1)
plt.plot(np.linspace(0, 1), np.linspace(0, 1) * 0.9 + 0.3, '--',␣
 ↪label='Separating line (example)')
plt.grid(which='both', axis='both', alpha=0.7, zorder=1)
plt.ylabel('y')
plt.xlabel('x')
plt.legend()
plt.title('Point position and labels in Dataset n.1')
plt.tight_layout()
plt.colorbar(label='Weather')
plt.show()
```



For dataset n.1, we can ideally draw a line to separate the points into two separate sets according to their labels. This is however not possible for the points in dataset n.2, as showed as follows.

```
[ ]: x_coords = []
y_coords = []
for x2 in X2:
    x_coords.append(x2['city'])
    y_coords.append(x2['month'])
```

```
plt.scatter(x_coords, y_coords, c=Y2)
plt.grid(which='both', axis='both', alpha=0.7, zorder=1)
plt.plot(np.linspace(0, 2), np.linspace(0, 1) * 0.55 + 0.1, '--',␣
 ↪label='Separating line (example)')
plt.ylabel('y')
plt.xlabel('x')
plt.title('Point position and labels in Dataset n.2')
plt.legend()
plt.tight_layout()
plt.colorbar(label='Weather')
plt.show()
```



As we can see, there is no straight line which can clearly separate the two sets. Hence, at least one point per label will be missclassified, leading to a lower accuracy score (in our case, 50%).

1.3 Introduction: Perceptron

I'll start by importing the provided Perceptron class.

[ ]:
```
import numpy as np
from sklearn.base import BaseEstimator

class LinearClassifier(BaseEstimator):
    """
    General class for binary linear classifiers. Implements the predict
    function, which is the same for all binary linear classifiers. There are
    also two utility functions.
    """

    def decision_function(self, X):
        """
        Computes the decision function for the inputs X. The inputs are assumed
        to be stored in a matrix, where each row contains the features for one
        instance.
        """
        return X.dot(self.w)

    def predict(self, X):
        """
        Predicts the outputs for the inputs X. The inputs are assumed to be
        stored in a matrix, where each row contains the features for one
        instance.
        """
        # First compute the output scores
        scores = self.decision_function(X)
        # Select the positive or negative class label, depending on whether
        # the score was positive or negative.
        out = np.select([scores >= 0.0, scores < 0.0],
                        [self.positive_class,
                         self.negative_class])
        return out

    def find_classes(self, Y):
        """
        Finds the set of output classes in the output part Y of the training
        set. If there are exactly two classes, one of them is associated to
        positive classifier scores, the other one to negative scores. If the
        number of classes is not 2, an error is raised.
        """
        classes = sorted(set(Y))
        if len(classes) != 2:
            raise Exception("this does not seem to be a 2-class problem")
        self.positive_class = classes[1]
        self.negative_class = classes[0]

    def encode_outputs(self, Y):
        """
        A helper function that converts all outputs to +1 or -1.
        """
        return np.array([1 if y == self.positive_class else -1 for y in Y])
```

```
class Perceptron(LinearClassifier):
    """
    A straightforward implementation of the perceptron learning algorithm.
    """
    def __init__(self, n_iter=20):
        """
        The constructor can optionally take a parameter n_iter specifying how
        many times we want to iterate through the training set.
        """
        self.n_iter = n_iter

    def fit(self, X, Y):
        """
        Train a linear classifier using the perceptron learning algorithm.
        """
        # First determine which output class will be associated with positive
        # and negative scores, respectively.
        self.find_classes(Y)
        # Convert all outputs to +1 (for the positive class) or -1 (negative).
        Ye = self.encode_outputs(Y)
        # If necessary, convert the sparse matrix returned by a vectorizer
        # into a normal NumPy matrix.
        if not isinstance(X, np.ndarray):
            X = X.toarray()
        # Initialize the weight vector to all zeros.
        n_features = X.shape[1]
        self.w = np.zeros(n_features)
        # Perceptron algorithm:
        for i in range(self.n_iter):
            for x, y in zip(X, Ye):
                # Compute the output score for this instance.
                score = x.dot(self.w)
                # If there was an error, update the weights.
                if y * score <= 0:
                    self.w += y * x
```

The custom Perceptron class can now be included into a data pipeline and tested.

[ ]:
```
import time

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.preprocessing import Normalizer
from sklearn.pipeline import make_pipeline
from sklearn.feature_selection import SelectKBest
from sklearn.metrics import accuracy_score
```

```python
from sklearn.model_selection import train_test_split

# This function reads the corpus, returns a list of documents, and a list
# of their corresponding polarity labels.
def read_data(corpus_file):
    X = []
    Y = []
    with open(corpus_file, encoding='utf-8') as f:
        for line in f:
            _, y, _, x = line.split(maxsplit=3)
            X.append(x.strip())
            Y.append(y)
    return X, Y

# Read all the documents.
X, Y = read_data(os.path.join(data_dir, 'all_sentiment_shuffled.txt'))
# Split into training and test parts.
Xtrain, Xtest, Ytrain, Ytest = train_test_split(X, Y, test_size=0.2,
                                                random_state=0)
# Set up the preprocessing steps and the classifier.
pipeline = make_pipeline(
    TfidfVectorizer(),
    SelectKBest(k=1000),
    Normalizer(),
    # NB that this is our Perceptron, not sklearn.linear_model.Perceptron
    Perceptron(n_iter=16)
)
# Train the classifier.
t0 = time.time()
pipeline.fit(Xtrain, Ytrain)
t1 = time.time()
print(f'Perceptron training time: {t1 - t0:.2f}s')
# Evaluate on the test set.
Yguess = pipeline.predict(Xtest)
accuracy = accuracy_score(Ytest, Yguess)
print(f'Perceptron accuracy: {accuracy:.4f}')
# Save metrics for comparison
models = {
    'Perceptron' : {
        'model': pipeline,
        'training_time': t1-t0,
        'accuracy': accuracy
    },
}
```

```
Perceptron training time: 2.61s
Perceptron accuracy: 0.7935
```

1.4 Implementing the SVC

The SVC `fit()` function shall implement the *hinge loss* for updating the weight parameter w.

[ ]:
```python
class LinearSVC(LinearClassifier):
    """
    A straightforward implementation of the linear SVC learning algorithm.
    """
    def __init__(self, n_iter=20, C=1):
        """
        The constructor can optionally take a parameter n_iter specifying how
        many times we want to iterate through the training set.
        """
        self.n_iter = n_iter
        self.C = C

    def fit(self, X, Y):
        """
        Train a linear classifier using the linear SVC learning algorithm.
        """
        # First determine which output class will be associated with positive
        # and negative scores, respectively.
        self.find_classes(Y)
        # Convert all outputs to +1 (for the positive class) or -1 (negative).
        Ye = self.encode_outputs(Y)
        # If necessary, convert the sparse matrix returned by a vectorizer
        # into a normal NumPy matrix.
        if not isinstance(X, np.ndarray):
            X = X.toarray()
        # Initialize the weight vector to all zeros.
        n_features = X.shape[1]
        self.w = np.zeros(n_features)
        t = 0
        XY = list(zip(X, Ye))
        for i in range(self.n_iter):
            # NOTE: The shuffling of the trainiing data can be removed, as it
            # causes slowdowns.
            p = np.random.permutation(X.shape[0])
            XY = list(zip(X[p], Ye[p]))
            for x, y in XY:
                t += 1
                eta = 1 / (self.C * t)
                # Compute the output score for this instance.
                score = x.dot(self.w)
                self.w = (1 - eta * self.C) * self.w
                if y * score < 1:
                    self.w += (eta * y) * x
```

Once that the class is implemented, it can be tested in a data pipeline as in the previous case.

```
[ ]: # Set up the preprocessing steps and the classifier.
svc_pipeline = make_pipeline(
    TfidfVectorizer(),
    SelectKBest(k=5000),
    Normalizer(),
    # NB that this is our LinearSVC, not sklearn.linear_model.LinearSVC
    LinearSVC(n_iter=16, C=1/len(Xtrain))
)
# Train the classifier.
t0 = time.time()
svc_pipeline.fit(Xtrain, Ytrain)
t1 = time.time()
print(f'SVC training time: {t1 - t0:.2f}s')
# Evaluate on the test set.
Yguess = svc_pipeline.predict(Xtest)
accuracy = accuracy_score(Ytest, Yguess)
print(f'SVC accuracy: {accuracy:.4f}')
models['LinearSVC'] = {
    'model': svc_pipeline,
    'training_time': t1-t0,
    'accuracy': accuracy
}
```

```
SVC training time: 8.46s
SVC accuracy: 0.8481
```

1.5 Logistic regression

For implementing a Logistic Regression classifier, the `fit()` function shall include the implementation of a *log loss*. In particular, for each training point, the weight vector w is updated according to the *learning rate* η and the loss gradient:

$\nabla(Loss) = -\frac{y_i}{1+exp(y_i \cdot w \cdot x_i)} \cdot x_i$

```
[ ]: class LogisticRegression(LinearClassifier):
    """
    A straightforward implementation of the perceptron learning algorithm.
    """

    def __init__(self, n_iter=20, C=1):
        """
        The constructor can optionally take a parameter n_iter specifying how
        many times we want to iterate through the training set.
        """
        self.n_iter = n_iter
        self.C = C

    def fit(self, X, Y):
        """
        Train a linear classifier using the perceptron learning algorithm.
        """
        # First determine which output class will be associated with positive
        # and negative scores, respectively.
        self.find_classes(Y)
        # Convert all outputs to +1 (for the positive class) or -1 (negative).
        Ye = self.encode_outputs(Y)
        # If necessary, convert the sparse matrix returned by a vectorizer
        # into a normal NumPy matrix.
        if not isinstance(X, np.ndarray):
            X = X.toarray()
        # Initialize the weight vector to all zeros.
        n_features = X.shape[1]
        self.w = np.zeros(n_features)
        t = 0
        XY = list(zip(X, Ye))
        for i in range(self.n_iter):
            # NOTE: The shuffling of the trainiing data can be removed, as it
            # causes slowdowns.
            p = np.random.permutation(X.shape[0])
            XY = list(zip(X[p], Ye[p]))
            for x, y in XY:
                t += 1
                eta = 1 / (self.C * t)
                wx = x.dot(self.w)
                self.w = (1 - eta * self.C) * self.w + y / (1 + np.exp(y * wx))␣
↪* x
```

The custom classifier can be included into a pipeline too. The `C`, *i.e.* hyperparameter can be tuned to achieve an accuracy score above 80%.

```
[ ]: # Set up the preprocessing steps and the classifier.
lrclassifier_pipeline = make_pipeline(
    TfidfVectorizer(),
    SelectKBest(k=1000),
    Normalizer(),
    # NB that this is our LogisticRegression, not sklearn.linear_model.
↪LogisticRegression
    LogisticRegression(n_iter=16, C=1/len(Xtrain))
)
# Train the classifier.
t0 = time.time()
lrclassifier_pipeline.fit(Xtrain, Ytrain)
```

```
t1 = time.time()
print(f'Logistic Regression training time: {t1 - t0:.2f}s')
# Evaluate on the test set.
Yguess = lrclassifier_pipeline.predict(Xtest)
accuracy = accuracy_score(Ytest, Yguess)
print(f'Logistic Regression accuracy: {accuracy:.4f}')
models['LogisticRegression'] = {
    'model': lrclassifier_pipeline,
    'training_time': t1-t0,
    'accuracy': accuracy
}
```

```
Logistic Regression training time: 4.78s
Logistic Regression accuracy: 0.8229
```

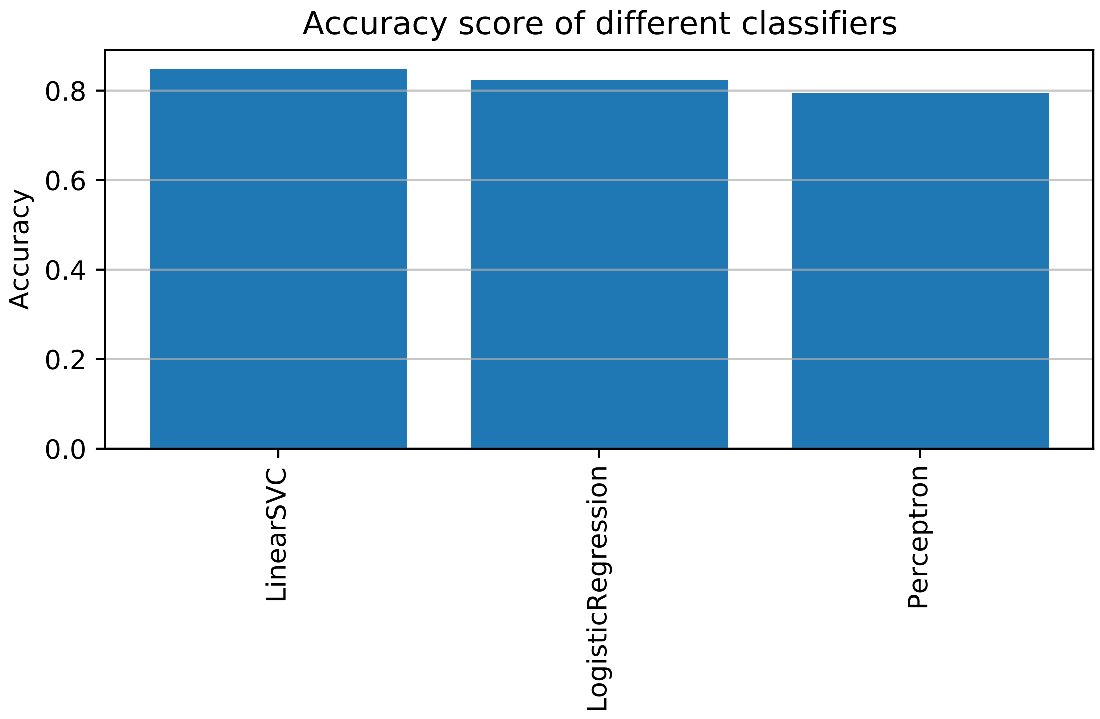
1.6 Comparison

Once all the classifiers have been trained, they can be compared in terms of accuracy and training time.

[ ]:
```
import matplotlib.pyplot as plt

def plot_accuracy(model_focus=''):
    scores = []
    models_keys = []
    for model_type in models.keys():
        if model_focus in model_type:
            models_keys.append(model_type)
            print(f'{model_type} accuracy: {models[model_type]["accuracy"]:.
 ↪3f}s')
            scores.append(models[model_type]['accuracy'])
    # Alphabetically sort scores according to the model types
    scores = [x for _, x in sorted(zip(models_keys, scores))]
    linspace = [x for x in range(len(models_keys))]
    plt.bar(linspace, scores)
    plt.xticks(linspace, [f'{m}' for m in sorted(models_keys)], rotation=90)
    plt.grid(which='both', axis='y', alpha=0.7, zorder=1)
    plt.ylabel('Accuracy')
    plt.title('Accuracy score of different classifiers')
    plt.tight_layout()
    # plt.savefig(os.path.join(data_dir, f'accuracy.pdf'))
    plt.show()

plot_accuracy()
```

```
Perceptron accuracy: 0.794s
LinearSVC accuracy: 0.848s
```

```
LogisticRegression accuracy: 0.823s
```



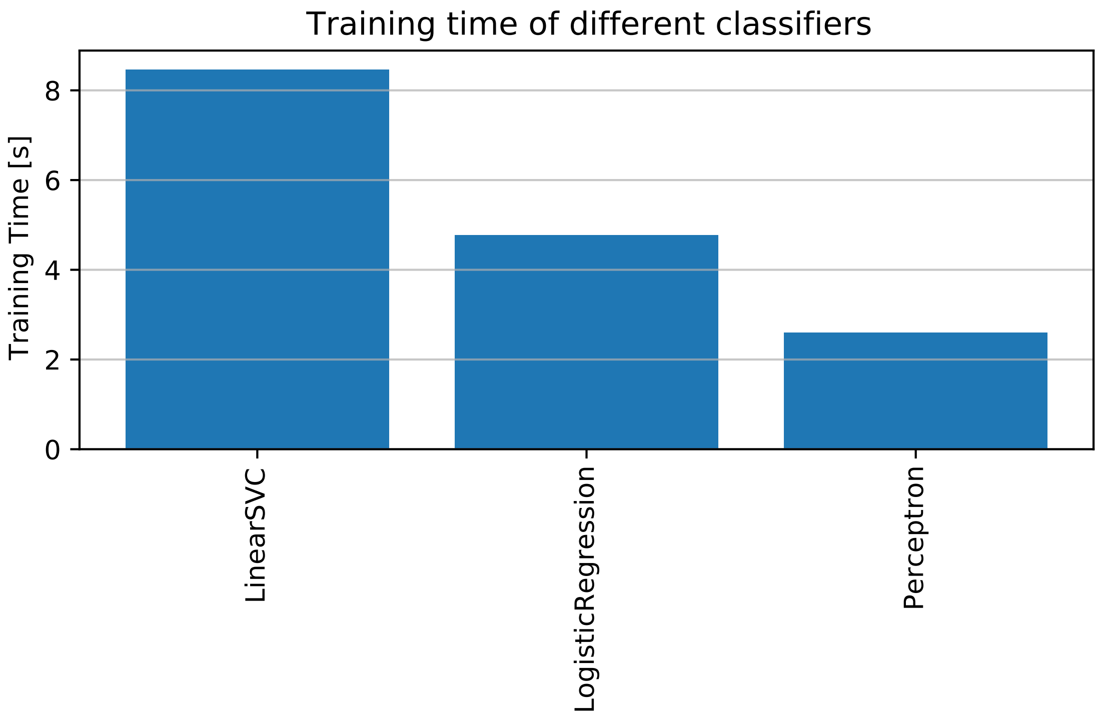
As it can the be seen in the graph above, compared to the Perceptron, both the SVC and Logistic Regression (LR) classifiers perform better in terms of accuracy.

The reason for that might lie in the Perceptron training algorithm, which mearly updates the w vector with the training data x_i numerical values. In contrast, the SVC aims instead at finding the best w which **minimizes** the SVC loss function.

```
[ ]: def plot_training_time(model_focus=''):
         scores = []
         models_keys = []
         for model_type in models.keys():
             if model_focus in model_type:
                 models_keys.append(model_type)
                 print(f'{model_type} training time:␣
      ↪{models[model_type]["training_time"]:.3f}s')
                 scores.append(models[model_type]['training_time'])
         # Alphabetically sort scores according to the model types
         scores = [x for _, x in sorted(zip(models_keys, scores))]
         linspace = [x for x in range(len(models_keys))]
         plt.bar(linspace, scores)
         plt.xticks(linspace, [f'{m}' for m in sorted(models_keys)], rotation=90)
         plt.grid(which='both', axis='y', alpha=0.7, zorder=1)
         plt.ylabel('Training Time [s]')
```

```
    plt.title('Training time of different classifiers')
    plt.tight_layout()
    # plt.savefig(os.path.join(data_dir, f'accuracy.pdf'))
    plt.show()

plot_training_time()
```

```
Perceptron training time: 2.605s
LinearSVC training time: 8.464s
LogisticRegression training time: 4.778s
```



However, because of its simplicity, the Perceptron classifier is way faster to train, requiring only 2.7s against the 8.5s of the SVC classifier (around 3.25× slower).

1.7 Bonus task 1: Making the code more efficient

1.7.1 (a) Faster linear algebra operations

```
[ ]: from scipy.linalg import blas
```

BLAS SVC

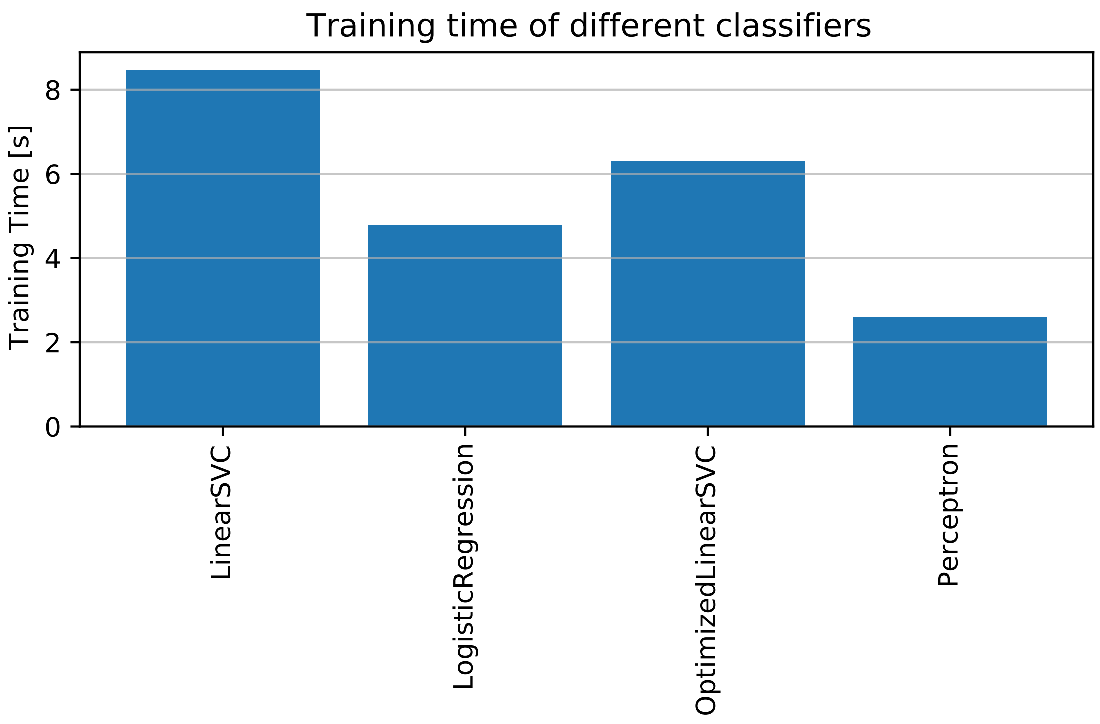
[ ]:
```
class OptimizedLinearSVC(LinearClassifier):
    """
    A straightforward implementation of the linear SVC learning algorithm.
    """
    def __init__(self, n_iter=20, C=1):
        """
        The constructor can optionally take a parameter n_iter specifying how
        many times we want to iterate through the training set.
        """
        self.n_iter = n_iter
        self.C = C

    def fit(self, X, Y):
        """
        Train a linear classifier using the linear SVC learning algorithm.
        """
        # First determine which output class will be associated with positive
        # and negative scores, respectively.
        self.find_classes(Y)
        # Convert all outputs to +1 (for the positive class) or -1 (negative).
        Ye = self.encode_outputs(Y)
        # If necessary, convert the sparse matrix returned by a vectorizer
        # into a normal NumPy matrix.
        if not isinstance(X, np.ndarray):
            X = X.toarray()
        # Initialize the weight vector to all zeros.
        n_features = X.shape[1]
        self.w = np.zeros(n_features)
        t = 0
        XY = list(zip(X, Ye))
        for i in range(self.n_iter):
            # NOTE: The shuffling of the trainiing data can be removed, as it
            # causes slowdowns.
            p = np.random.permutation(X.shape[0])
            XY = list(zip(X[p], Ye[p]))
            for x, y in XY:
                t += 1
                eta = 1 / (self.C * t)
                # Compute the output score for this instance.
                score = blas.ddot(x, self.w)
                self.w = blas.dscal((1 - eta * self.C), self.w)
                if y * score < 1:
                    blas.daxpy(x, self.w, a=eta * y)
```

[ ]:
```
# Set up the preprocessing steps and the classifier.
svc_pipeline_opt = make_pipeline(
    TfidfVectorizer(),
    SelectKBest(k=5000),
    Normalizer(),
    OptimizedLinearSVC(n_iter=16, C=1/len(Xtrain))
)
# Train the classifier.
t0 = time.time()
svc_pipeline_opt.fit(Xtrain, Ytrain)
t1 = time.time()
print(f'Optimized SVC training time: {t1 - t0:.2f}s')
# Evaluate on the test set.
Yguess = svc_pipeline_opt.predict(Xtest)
accuracy = accuracy_score(Ytest, Yguess)
print(f'Optimized SVC accuracy: {accuracy:.4f}')
models['OptimizedLinearSVC'] = {
    'model': svc_pipeline_opt,
    'training_time': t1-t0,
    'accuracy': accuracy
}
```

```
Optimized SVC training time: 6.32s
Optimized SVC accuracy: 0.8468
```

[ ]:
```
plot_training_time()
```

```
Perceptron training time: 2.605s
LinearSVC training time: 8.464s
LogisticRegression training time: 4.778s
OptimizedLinearSVC training time: 6.316s
```



As seen from the comparison above, when using BLAS functions we see a speedup of 1.34× for the SVC classifier. The accuracy score is uneffected.

BLAS Logistic Regression

```
[ ]: class OptimizedLogisticRegression(LinearClassifier):
         """
         A straightforward implementation of the perceptron learning algorithm.
         """

         def __init__(self, n_iter=20, C=1):
             """
             The constructor can optionally take a parameter n_iter specifying how
             many times we want to iterate through the training set.
             """
             self.n_iter = n_iter
             self.C = C

         def fit(self, X, Y):
             """
             Train a linear classifier using the perceptron learning algorithm.
             """
             # First determine which output class will be associated with positive
             # and negative scores, respectively.
```

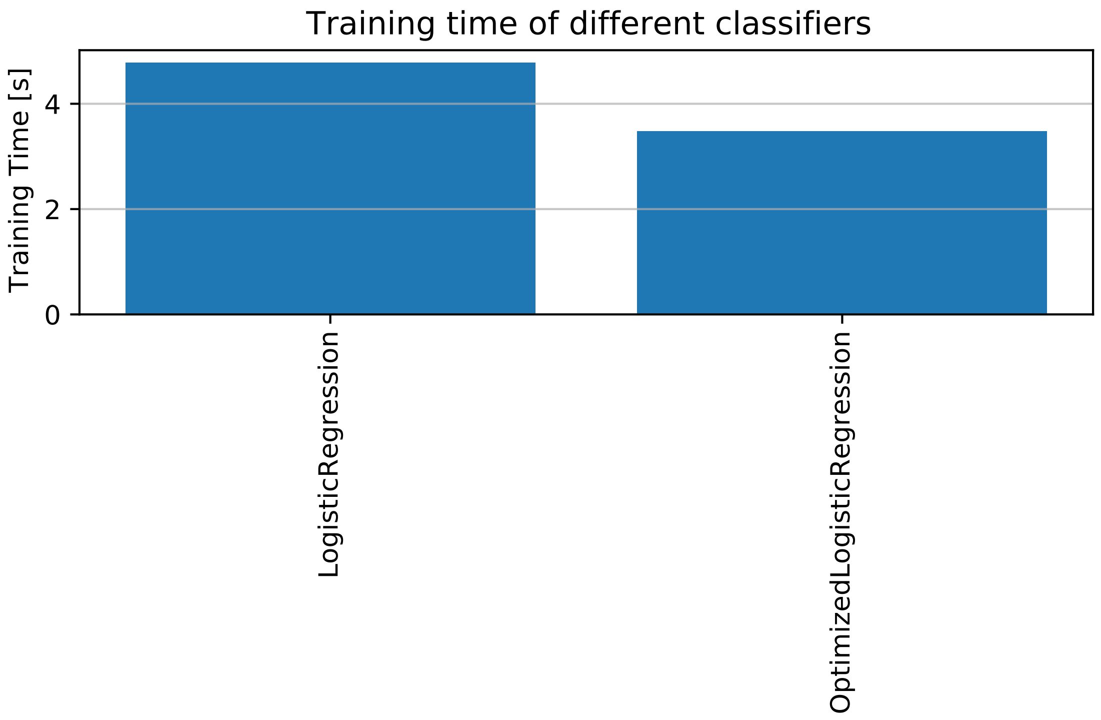
```
        self.find_classes(Y)
        # Convert all outputs to +1 (for the positive class) or -1 (negative).
        Ye = self.encode_outputs(Y)
        # If necessary, convert the sparse matrix returned by a vectorizer
        # into a normal NumPy matrix.
        if not isinstance(X, np.ndarray):
            X = X.toarray()
        # Initialize the weight vector to all zeros.
        n_features = X.shape[1]
        self.w = np.zeros(n_features)
        t = 0
        XY = list(zip(X, Ye))
        for i in range(self.n_iter):
            # NOTE: The shuffling of the trainiing data can be removed, as it
            # causes slowdowns.
            p = np.random.permutation(X.shape[0])
            XY = list(zip(X[p], Ye[p]))
            for x, y in XY:
                t += 1
                # NOTE: eta * C = 1 / t
                eta = 1 / (self.C * t)
                self.w = blas.dscal(1 - 1 / t, self.w)
                wx = blas.ddot(x, self.w)
                ywx = blas.dscal(y, wx)
                blas.daxpy(x, self.w, a=y / (1 + np.exp(ywx)))
```

[ ]:
```
# Set up the preprocessing steps and the classifier.
lrclassifier_pipeline = make_pipeline(
    TfidfVectorizer(),
    SelectKBest(k=1000),
    Normalizer(),
    OptimizedLogisticRegression(n_iter=16, C=1/len(Xtrain))
)
# Train the classifier.
t0 = time.time()
lrclassifier_pipeline.fit(Xtrain, Ytrain)
t1 = time.time()
print(f'Optimized Logistic Regression training time: {t1 - t0:.2f}s')
# Evaluate on the test set.
Yguess = lrclassifier_pipeline.predict(Xtest)
accuracy = accuracy_score(Ytest, Yguess)
print(f'Optimized Logistic Regression accuracy: {accuracy:.4f}')
models['OptimizedLogisticRegression'] = {
    'model': lrclassifier_pipeline,
    'training_time': t1-t0,
    'accuracy': accuracy
}
```

```
Optimized Logistic Regression training time: 3.48s
Optimized Logistic Regression accuracy: 0.8347
```

```
[ ]: plot_training_time(model_focus='LogisticRegression')
```

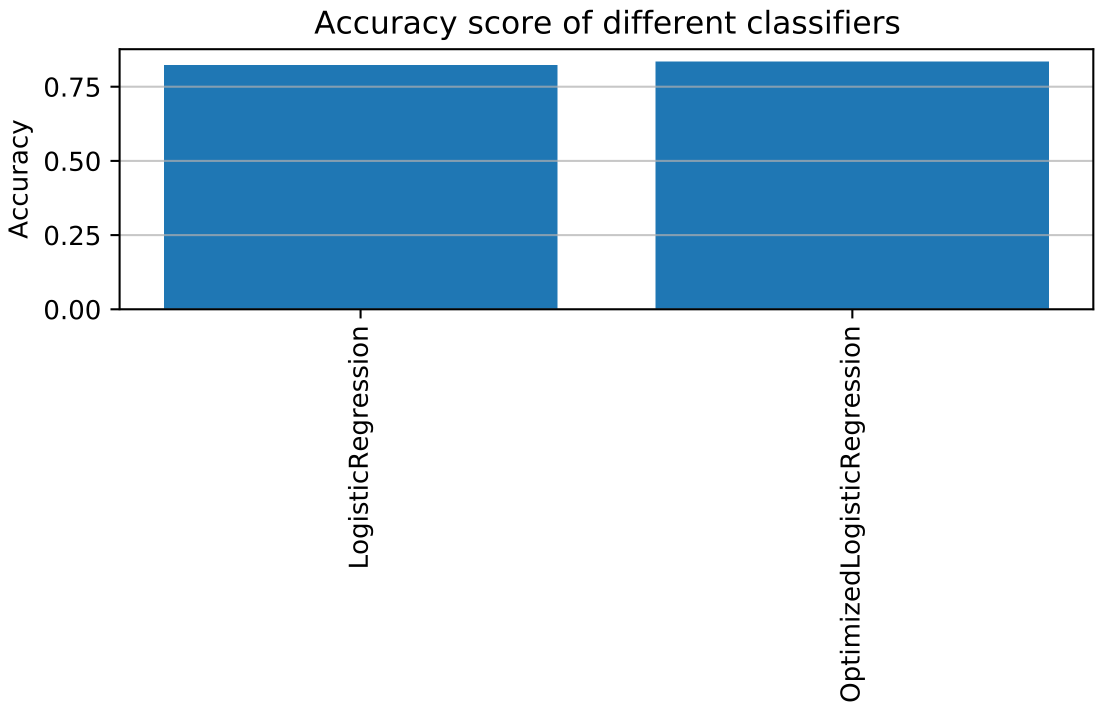
```
LogisticRegression training time: 4.778s
OptimizedLogisticRegression training time: 3.483s
```



We see a comparable speedup when optimizing the LR classifier: 1.37×. Again, the accuracy score is not changing.

```
[ ]: plot_accuracy(model_focus='LogisticRegression')
```

```
LogisticRegression accuracy: 0.823s
OptimizedLogisticRegression accuracy: 0.835s
```



1.7.2 (b) Using sparse vectors

In the following experiments, I've removed the `SelectKBest(k=5000)` element from the pipeline, as it was limiting the total number of features to utilize (and in turn, it determined the maximum length of the input vectors).

I decided to limit my analysis to a Vectorizer considering only unigrams. This was based on the fact that training using unigrams only was already comparatively slower than the baseline case and with high accuracy score. Hence, I believe that using both uni- and bi-grams would increase the number of features and lead to similar speedups and slowdown results.

Sparse Perceptron

```
[ ]: ### Sparse and dense vectors don't collaborate very well in NumPy/SciPy.
     ### Here are two utility functions that help us carry out some vector
     ### operations that we'll need.

     def add_sparse_to_dense(x, w, factor):
         """
         Adds a sparse vector x, scaled by some factor, to a dense vector.
         This can be seen as the equivalent of w += factor * x when x is a dense
         vector.
         """
```

```python
    w[x.indices] += factor * x.data

def sparse_dense_dot(x, w):
    """
    Computes the dot product between a sparse vector x and a dense vector w.
    """
    return np.dot(w[x.indices], x.data)


class SparsePerceptron(LinearClassifier):
    """
    A straightforward implementation of the perceptron learning algorithm,
    assuming that the input feature matrix X is sparse.
    """
    def __init__(self, n_iter=20):
        """
        The constructor can optionally take a parameter n_iter specifying how
        many times we want to iterate through the training set.
        """
        self.n_iter = n_iter

    def fit(self, X, Y):
        """
        Train a linear classifier using the perceptron learning algorithm.

        Note that this will only work if X is a sparse matrix, such as the
        output of a scikit-learn vectorizer.
        """
        self.find_classes(Y)
        # First determine which output class will be associated with positive
        # and negative scores, respectively.
        Ye = self.encode_outputs(Y)
        # Initialize the weight vector to all zeros.
        self.w = np.zeros(X.shape[1])
        # Iteration through sparse matrices can be a bit slow, so we first
        # prepare this list to speed up iteration.
        XY = list(zip(X, Ye))
        for i in range(self.n_iter):
            for x, y in XY:
                # Compute the output score for this instance.
                # (This corresponds to score = x.dot(self.w) above.)
                score = sparse_dense_dot(x, self.w)
                # If there was an error, update the weights.
                if y * score <= 0:
                    # (This corresponds to self.w += y*x above.)
                    add_sparse_to_dense(x, self.w, y)
```

```
[ ]: # Set up the preprocessing steps and the classifier.
     pipeline = make_pipeline(
         TfidfVectorizer(),
         Normalizer(),
         SparsePerceptron(n_iter=16)
     )
     # Train the classifier.
     t0 = time.time()
     pipeline.fit(Xtrain, Ytrain)
     t1 = time.time()
     print(f'PerceptronMaxFeatures training time: {t1 - t0:.2f}s')
     # Evaluate on the test set.
     Yguess = pipeline.predict(Xtest)
     accuracy = accuracy_score(Ytest, Yguess)
     print(f'PerceptronMaxFeatures accuracy: {accuracy:.4f}')
     # Save metrics for comparison
     models['PerceptronMaxFeatures'] = {
         'model': pipeline,
         'training_time': t1-t0,
         'accuracy': accuracy
     }
```

```
PerceptronMaxFeatures training time: 7.16s
PerceptronMaxFeatures accuracy: 0.8279
```

```
[ ]: # Set up the preprocessing steps and the classifier.
     pipeline = make_pipeline(
         TfidfVectorizer(),
         SelectKBest(k=1000),
         Normalizer(),
         SparsePerceptron(n_iter=16)
     )
     # Train the classifier.
     t0 = time.time()
     pipeline.fit(Xtrain, Ytrain)
     t1 = time.time()
     print(f'SparsePerceptron training time: {t1 - t0:.2f}s')
     # Evaluate on the test set.
     Yguess = pipeline.predict(Xtest)
     accuracy = accuracy_score(Ytest, Yguess)
     print(f'SparsePerceptron accuracy: {accuracy:.4f}')
     # Save metrics for comparison
     models['SparsePerceptron'] = {
         'model': pipeline,
         'training_time': t1-t0,
         'accuracy': accuracy
     }
```

```
SparsePerceptron training time: 7.10s
SparsePerceptron accuracy: 0.7935
```

```
[ ]: # Set up the preprocessing steps and the classifier.
     pipeline = make_pipeline(
         TfidfVectorizer(),
         Normalizer(),
         SparsePerceptron(n_iter=16)
     )
     # Train the classifier.
     t0 = time.time()
     pipeline.fit(Xtrain, Ytrain)
     t1 = time.time()
     print(f'SparsePerceptron training time: {t1 - t0:.2f}s')
     # Evaluate on the test set.
     Yguess = pipeline.predict(Xtest)
     accuracy = accuracy_score(Ytest, Yguess)
     print(f'SparsePerceptron accuracy: {accuracy:.4f}')
     # Save metrics for comparison
     models['SparsePerceptronMaxFeatures'] = {
         'model': pipeline,
         'training_time': t1-t0,
         'accuracy': accuracy
     }
```

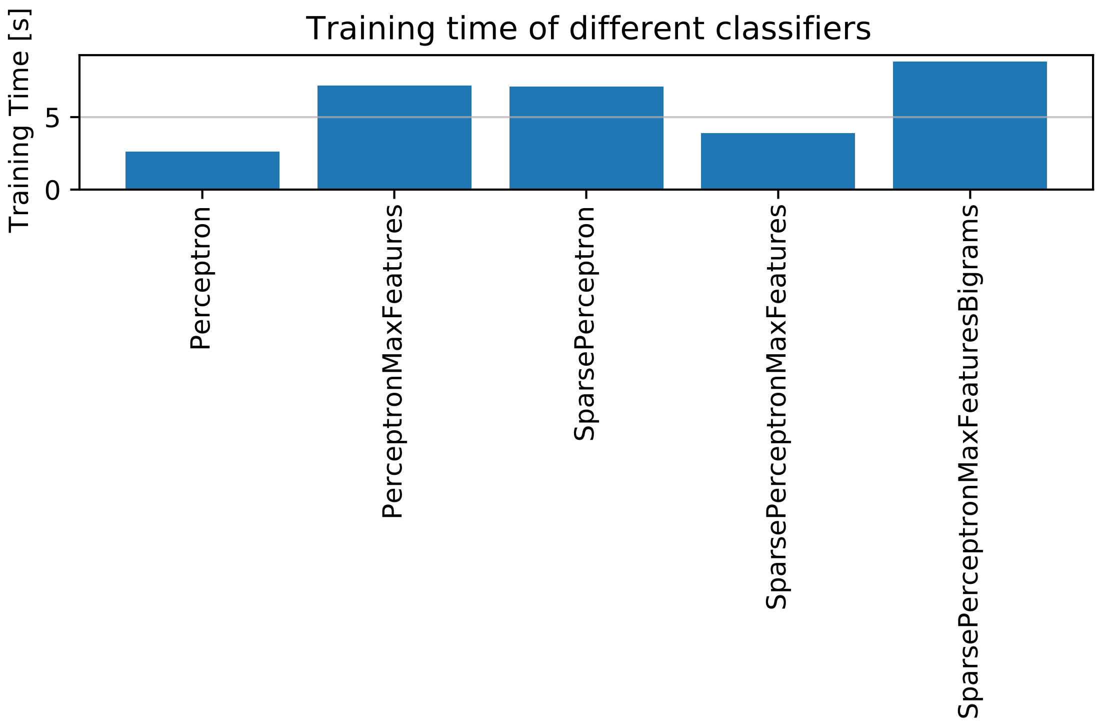
```
SparsePerceptron training time: 3.89s
SparsePerceptron accuracy: 0.8279
```

```
[ ]: # Set up the preprocessing steps and the classifier.
     pipeline = make_pipeline(
         TfidfVectorizer(ngram_range=(1,2)),
         Normalizer(),
         SparsePerceptron(n_iter=16)
     )
     # Train the classifier.
     t0 = time.time()
     pipeline.fit(Xtrain, Ytrain)
     t1 = time.time()
     print(f'SparsePerceptron training time: {t1 - t0:.2f}s')
     # Evaluate on the test set.
     Yguess = pipeline.predict(Xtest)
     accuracy = accuracy_score(Ytest, Yguess)
     print(f'SparsePerceptron accuracy: {accuracy:.4f}')
     # Save metrics for comparison
     models['SparsePerceptronMaxFeaturesBigrams'] = {
         'model': pipeline,
         'training_time': t1-t0,
         'accuracy': accuracy
     }
```

```
SparsePerceptron training time: 8.83s
SparsePerceptron accuracy: 0.8636
```

[ ]:
```
plot_training_time(model_focus='Perceptron')
```

```
Perceptron training time: 2.605s
PerceptronMaxFeatures training time: 7.162s
SparsePerceptron training time: 7.098s
SparsePerceptronMaxFeatures training time: 3.885s
SparsePerceptronMaxFeaturesBigrams training time: 8.832s
```



Sparse SVC

[ ]:
```
class SparseLinearSVC(LinearClassifier):
    """
    A straightforward implementation of the linear SVC learning algorithm.
    """
    def __init__(self, n_iter=20, C=1):
        """
        The constructor can optionally take a parameter n_iter specifying how
        many times we want to iterate through the training set.
        """
        self.n_iter = n_iter
        self.C = C

    def fit(self, X, Y):
        """
        Train a linear classifier using the linear SVC learning algorithm.
        """
        # First determine which output class will be associated with positive
        # and negative scores, respectively.
        self.find_classes(Y)
        # Convert all outputs to +1 (for the positive class) or -1 (negative).
        Ye = self.encode_outputs(Y)
        # Initialize the weight vector to all zeros.
        n_features = X.shape[1]
        self.w = np.zeros(n_features)
        t = 0
        # Iteration through sparse matrices can be a bit slow, so we first
        # prepare this list to speed up iteration.
        XY = list(zip(X, Ye))
        for i in range(self.n_iter):
            # NOTE: The shuffling of the traininig data was removed, as it
            # caused too much slowdown.
            # p = np.random.permutation(X.shape[0])
            # XY = list(zip(X[p], Ye[p]))
            for x, y in XY:
                t += 1
                eta = 1 / (self.C * t)
                # Compute the output score for this instance.
                score = sparse_dense_dot(x, self.w)
                self.w = (1 - eta * self.C) * self.w
                if y * score < 1:
                    add_sparse_to_dense(x, self.w, y * eta)
```

[ ]:
```
# Set up the preprocessing steps and the classifier.
svc_pipeline = make_pipeline(
    TfidfVectorizer(),
    Normalizer(),
    LinearSVC(n_iter=16, C=1/len(Xtrain))
)
# Train the classifier.
t0 = time.time()
svc_pipeline.fit(Xtrain, Ytrain)
t1 = time.time()
print(f'SVCMaxFeatures training time: {t1 - t0:.2f}s')
# Evaluate on the test set.
```

```
Yguess = svc_pipeline.predict(Xtest)
accuracy = accuracy_score(Ytest, Yguess)
print(f'SVCMaxFeatures accuracy: {accuracy:.4f}')
models['LinearSVCMaxFeatures'] = {
    'model': svc_pipeline,
    'training_time': t1-t0,
    'accuracy': accuracy
}
```

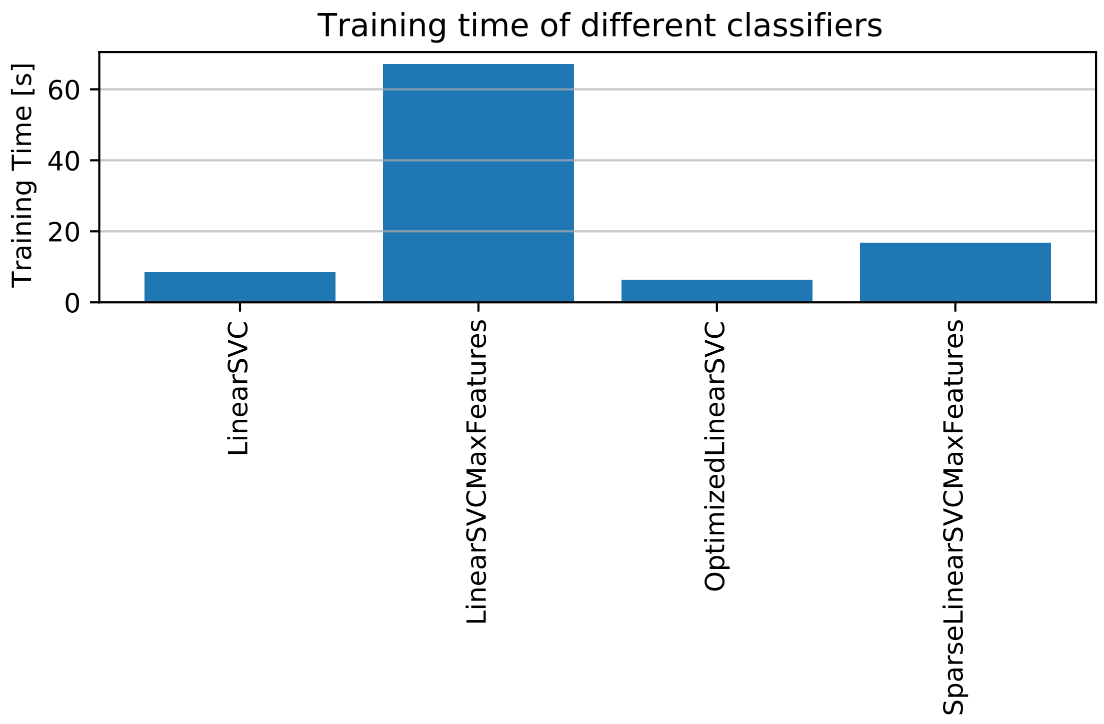
```
SVCMaxFeatures training time: 67.13s
SVCMaxFeatures accuracy: 0.8414
```

[ ]:
```
# Set up the preprocessing steps and the classifier.
svc_pipeline = make_pipeline(
    TfidfVectorizer(),
    Normalizer(),
    SparseLinearSVC(n_iter=16, C=1/len(Xtrain))
)
# Train the classifier.
t0 = time.time()
svc_pipeline.fit(Xtrain, Ytrain)
t1 = time.time()
print(f'SVC training time: {t1 - t0:.2f}s')
# Evaluate on the test set.
Yguess = svc_pipeline.predict(Xtest)
accuracy = accuracy_score(Ytest, Yguess)
print(f'SVC accuracy: {accuracy:.4f}')
models['SparseLinearSVCMaxFeatures'] = {
    'model': svc_pipeline,
    'training_time': t1-t0,
    'accuracy': accuracy
}
```

```
SVC training time: 16.71s
SVC accuracy: 0.8414
```

[ ]:
```
plot_training_time(model_focus='SVC')
```

```
LinearSVC training time: 8.464s
OptimizedLinearSVC training time: 6.316s
LinearSVCMaxFeatures training time: 67.126s
SparseLinearSVCMaxFeatures training time: 16.713s
```



We can see that when considering all available features (marked as ”MaxFeatures”), the training time of the unoptimized SVC classifier is more than 8× slower than the original case. On the other hand, the slowdown is only of 2× when sparse operations are utilized instead.

Sparse Logistic Regression

```
[ ]: class SparseLogisticRegression(LinearClassifier):
         """
         A straightforward implementation of the perceptron learning algorithm.
         """

         def __init__(self, n_iter=20, C=1):
             """
             The constructor can optionally take a parameter n_iter specifying how
             many times we want to iterate through the training set.
             """
             self.n_iter = n_iter
             self.C = C

         def fit(self, X, Y):
             """
             Train a linear classifier using the perceptron learning algorithm.
             """
             # First determine which output class will be associated with positive
```

```
        # and negative scores, respectively.
        self.find_classes(Y)
        # Convert all outputs to +1 (for the positive class) or -1 (negative).
        Ye = self.encode_outputs(Y)
        # Initialize the weight vector to all zeros.
        n_features = X.shape[1]
        self.w = np.zeros(n_features)
        t = 0
        # Iteration through sparse matrices can be a bit slow, so we first
        # prepare this list to speed up iteration.
        XY = list(zip(X, Ye))
        for i in range(self.n_iter):
            # NOTE: The shuffling of the trainiing data was removed, as it
            # caused too much slowdown.
            # p = np.random.permutation(X.shape[0])
            # XY = list(zip(X[p], Ye[p]))
            for x, y in XY:
                t += 1
                # NOTE: eta * C = 1 / t
                self.w = (1 - 1 / t) * self.w
                ywx = y * sparse_dense_dot(x, self.w)
                add_sparse_to_dense(x, self.w, y / (1 + np.exp(ywx)))
```

```
[ ]: # Set up the preprocessing steps and the classifier.
lrclassifier_pipeline = make_pipeline(
    TfidfVectorizer(),
    Normalizer(),
    LogisticRegression(n_iter=16, C=1/len(Xtrain))
)
# Train the classifier.
t0 = time.time()
lrclassifier_pipeline.fit(Xtrain, Ytrain)
t1 = time.time()
print(f'LogisticRegressionMaxFeatures training time: {t1 - t0:.2f}s')
# Evaluate on the test set.
Yguess = lrclassifier_pipeline.predict(Xtest)
accuracy = accuracy_score(Ytest, Yguess)
print(f'LogisticRegressionMaxFeatures accuracy: {accuracy:.4f}')
models['LogisticRegressionMaxFeatures'] = {
    'model': lrclassifier_pipeline,
    'training_time': t1-t0,
    'accuracy': accuracy
}
```

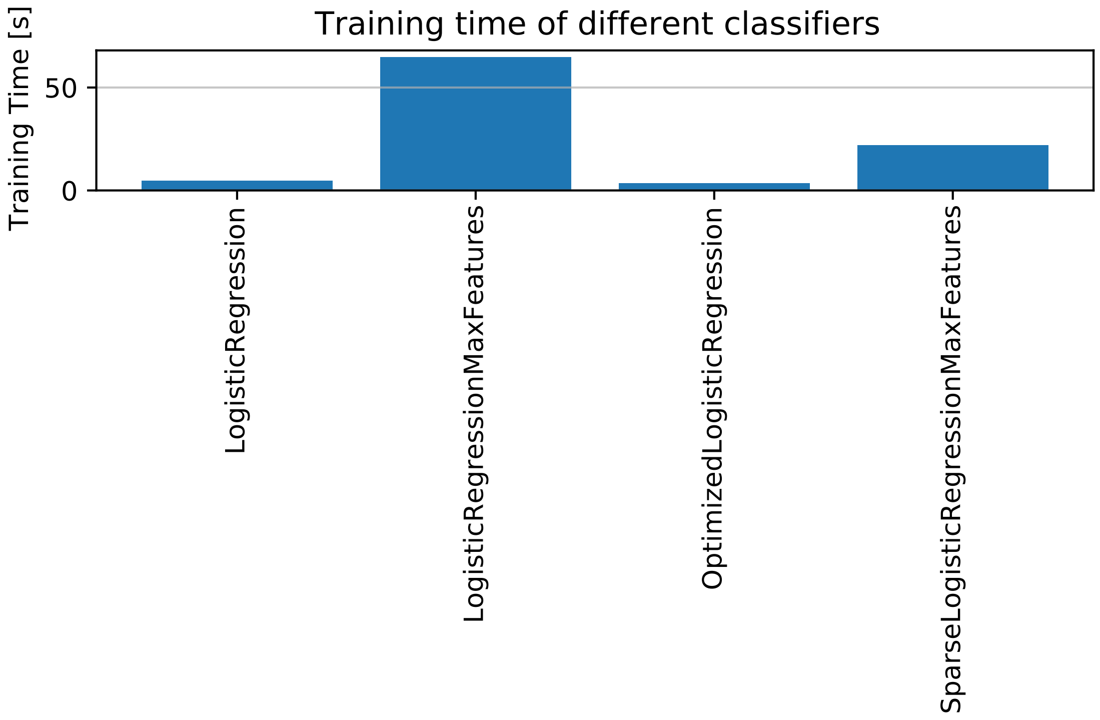
```
LogisticRegressionMaxFeatures training time: 64.79s
LogisticRegressionMaxFeatures accuracy: 0.8414
```

[ ]:
```
# Set up the preprocessing steps and the classifier.
lrclassifier_pipeline = make_pipeline(
    TfidfVectorizer(),
    Normalizer(),
    SparseLogisticRegression(n_iter=16, C=1/len(Xtrain))
)
# Train the classifier.
t0 = time.time()
lrclassifier_pipeline.fit(Xtrain, Ytrain)
t1 = time.time()
print(f'SparseLogisticRegressionMaxFeatures training time: {t1 - t0:.2f}s')
# Evaluate on the test set.
Yguess = lrclassifier_pipeline.predict(Xtest)
accuracy = accuracy_score(Ytest, Yguess)
print(f'SparseLogisticRegressionMaxFeatures accuracy: {accuracy:.4f}')
models['SparseLogisticRegressionMaxFeatures'] = {
    'model': lrclassifier_pipeline,
    'training_time': t1-t0,
    'accuracy': accuracy
}
```

```
SparseLogisticRegressionMaxFeatures training time: 22.04s
SparseLogisticRegressionMaxFeatures accuracy: 0.8422
```

[ ]:
```
plot_training_time(model_focus='LogisticRegression')
```

```
LogisticRegression training time: 4.778s
OptimizedLogisticRegression training time: 3.483s
LogisticRegressionMaxFeatures training time: 64.785s
SparseLogisticRegressionMaxFeatures training time: 22.038s
```

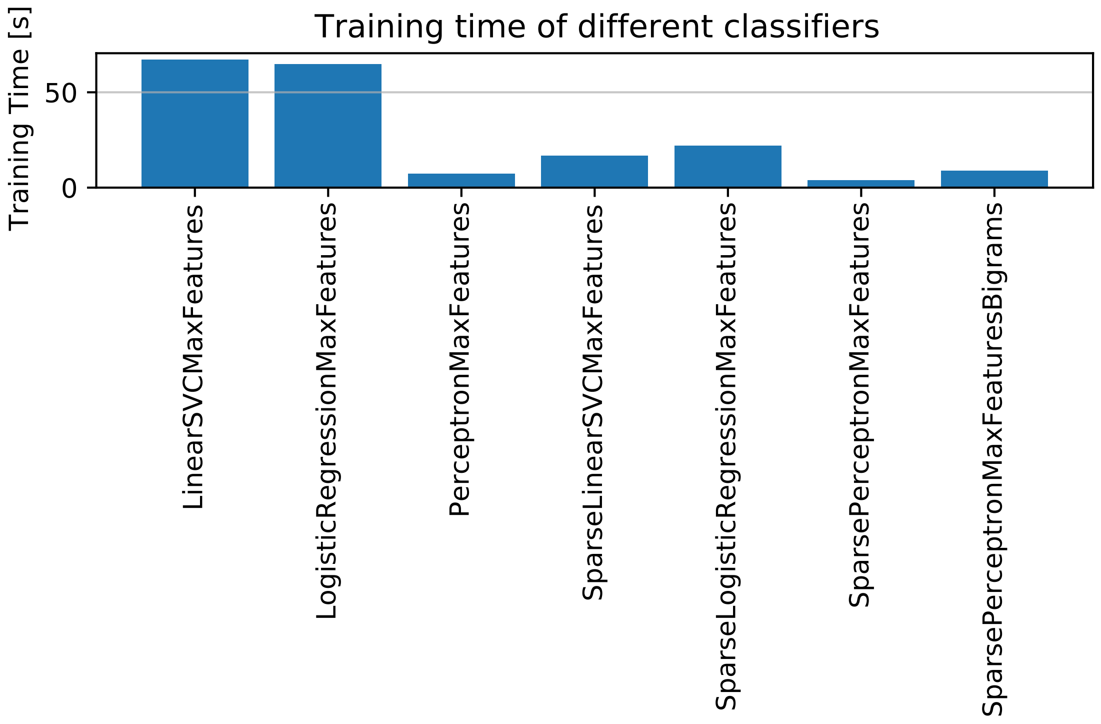


A similar comparison analysis can be made for the LR classifier. Without optimization, the training time is 13.6× longer than without using all available features. The training time of utilizing all features and sparse operations is instead of 4.6×.

Summary

```
[ ]: plot_training_time(model_focus='MaxFeatures')
```

```
PerceptronMaxFeatures training time: 7.162s
SparsePerceptronMaxFeatures training time: 3.885s
SparsePerceptronMaxFeaturesBigrams training time: 8.832s
LinearSVCMaxFeatures training time: 67.126s
SparseLinearSVCMaxFeatures training time: 16.713s
LogisticRegressionMaxFeatures training time: 64.785s
SparseLogisticRegressionMaxFeatures training time: 22.038s
```



1.7.3 (c) Speeding up the scaling operation

Opt. Scaling Logistic SVC

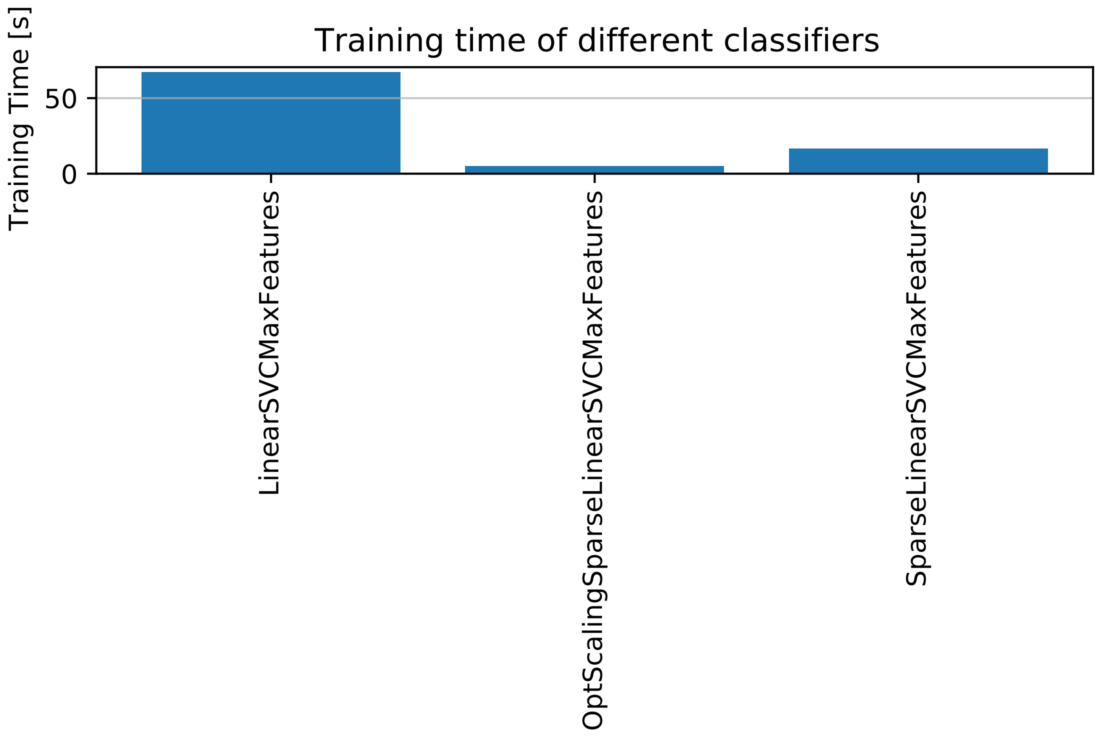
[ ]:
```
class OptScalingSparseLinearSVC(LinearClassifier):
    """
    A straightforward implementation of the linear SVC learning algorithm.
    """
    def __init__(self, n_iter=20, C=1):
        """
        The constructor can optionally take a parameter n_iter specifying how
        many times we want to iterate through the training set.
        """
        self.n_iter = n_iter
        self.C = C

    def fit(self, X, Y):
        """
        Train a linear classifier using the linear SVC learning algorithm.
        """
        # First determine which output class will be associated with positive
        # and negative scores, respectively.
        self.find_classes(Y)
        # Convert all outputs to +1 (for the positive class) or -1 (negative).
        Ye = self.encode_outputs(Y)
        # Initialize the weight vector to all zeros.
        n_features = X.shape[1]
        self.w = np.zeros(n_features)
        # NOTE: To avoid dividing by zero, we start from iteration t = 1
        t = 1
        # Iteration through sparse matrices can be a bit slow, so we first
        # prepare this list to speed up iteration.
        XY = list(zip(X, Ye))
        w_scaler = 1
        for i in range(self.n_iter):
            # NOTE: The shuffling of the trainiing data was removed, as it
            # caused too much slowdown.
            # p = np.random.permutation(X.shape[0])
            # XY = list(zip(X[p], Ye[p]))
            for x, y in XY:
                t += 1
                eta = 1 / (self.C * t)
                w_scaler = (1 - eta * self.C) * w_scaler
                score = w_scaler * sparse_dense_dot(x, self.w)
                if y * score < 1:
                    add_sparse_to_dense(x, self.w, eta * y / w_scaler)
        self.w = w_scaler * self.w
```

```
[ ]: # Set up the preprocessing steps and the classifier.
svc_pipeline = make_pipeline(
    TfidfVectorizer(),
    Normalizer(),
    OptScalingSparseLinearSVC(n_iter=16, C=1/len(Xtrain))
)
# Train the classifier.
t0 = time.time()
svc_pipeline.fit(Xtrain, Ytrain)
t1 = time.time()
print(f'OptScalingSparseLinearSVCMaxFeatures training time: {t1 - t0:.2f}s')
# Evaluate on the test set.
Yguess = svc_pipeline.predict(Xtest)
accuracy = accuracy_score(Ytest, Yguess)
print(f'OptScalingSparseLinearSVCMaxFeatures accuracy: {accuracy:.4f}')
models['OptScalingSparseLinearSVCMaxFeatures'] = {
    'model': svc_pipeline,
    'training_time': t1-t0,
    'accuracy': accuracy
}
```

```
OptScalingSparseLinearSVCMaxFeatures training time: 4.97s
OptScalingSparseLinearSVCMaxFeatures accuracy: 0.8401
```

```
[ ]: plot_training_time(model_focus='SVCMaxFeatures')
```

```
LinearSVCMaxFeatures training time: 67.126s
SparseLinearSVCMaxFeatures training time: 16.713s
OptScalingSparseLinearSVCMaxFeatures training time: 4.973s
```



As mentioned above, the training time of the unoptimized SVC classifier is more than 8× slower than the original case. The slowdown is only of 2× when sparse operations are utilized instead. Finally, when also reducing the scaling operations, *i.e.* applying the scaling ”trick”, the slowdown is of 0.59×. This means that by utilizing the combination of sparse operations and scaling trick, the training is actually 1.7× **faster** when using all available features than when using only a limited set of them (baseline case).

Opt. Scaling Logistic Regression (TODO).

2 Converting Notebook to PDF

The following two cells can be ignored for grading, as they just convert this notebook into a PDF file.

[3]:
```
%%capture
!apt-get update
!apt-get install -y texlive-xetex texlive-fonts-recommended␣
↪texlive-plain-generic
!apt-get install -y inkscape
!add-apt-repository -y universe
!add-apt-repository -y ppa:inkscape.dev/stable
!apt-get update -y
!apt install -y inkscape
```

[4]:
```
%%capture
import re

ASSIGNMENT_NAME = 'DAT340 - Assignment ' + ASSIGNMENT_ID.split('_')[1]
pdf_dir = os.path.join(os.path.abspath(''), 'drive', 'MyDrive')
pdf_dir = os.path.join(pdf_dir, 'Colab Notebooks', 'dat340', ASSIGNMENT_ID)
pdf_filename = re.escape(os.path.join(pdf_dir, ASSIGNMENT_NAME)) + '.ipynb'

!jupyter nbconvert --to pdf --TemplateExporter.exclude_input=False $pdf_filename
```

[ ]:
```

```